Programming Interfaces and Basic Topology

Peter Hancock[*] and Pierre Hyvernat[†]

VERSION of May 26, 2009 at 02:43

Abstract

A pattern of interaction that arises again and again in programming, is a “handshake”, in which two agents exchange data. The exchange is thought of as provision of a service. Each interaction is initiated by a specific agent —the client or Angel, and concluded by the other —the server or Demon.

We present a category in which the objects —called interaction structures in the paper— serve as descriptions of services provided across such handshaken interfaces. The morphisms —called (general) simulations— model components that provide one such service, relying on another. The morphisms are relations between the underlying sets of the interaction structures. The proof that a relation is a simulation can serve (in principle) as an executable program, whose specification is that it provides the service described by its domain, given an implementation of the service described by its codomain.

This category is then shown to coincide with the subcategory of “generated” basic topologies in Sambin’s terminology, where a basic topology is given by a closure operator whose induced sup-lattice structure need not be distributive; and moreover, this operator is inductively generated from a basic cover relation. This coincidence provides topologists with a natural source of examples for non-distributive formal topology. It raises a number of questions of interest both for formal topology and programming.

The extra structure needed to make such a basic topology into a real *formal topology* is then interpreted in the context of interaction structures.

[*] hancock@spamcop.net

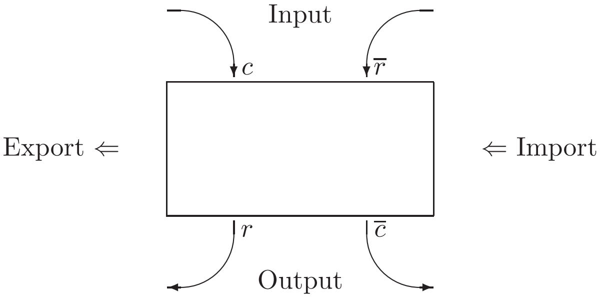
[†] hyvernat@iml.univ-mrs.fr

Contents

1 Introduction, preliminaries and notation

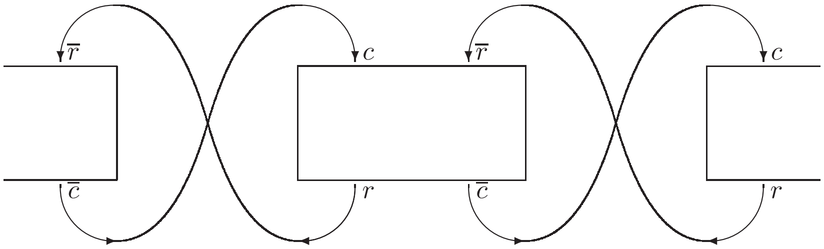
Programmers rarely write self-standing programs, but rather modules or components in a complete system. The boundaries of components are known as interfaces, and these usually take the form of collections of procedures. Commonly, a component exports or implements a "high-level" interface (for example files and directory trees in a file system) by making use of another "low-level" interface (for example segments of magnetic media on disk drives). There is, as it were, a conditional guarantee: the exported interface will work properly *provided* that the imported one works properly.

One picture for the programmer's task is therefore this:



The task is to "fill the box". In this picture the horizontal dimension shows interfaces. The exported, higher-level interface is at the left and the imported, lower-level interface at the right. The vertical dimension shows communication events (calls to and returns from procedures), with data flowing from top to bottom: c and $\overline{r}$ communicate data from the environment, while r and $\overline{c}$ communicate data to the environment. The labels c (for command or call) and r (for response or return) constitute events in the higher level interface, while $\overline{c}$ and $\overline{r}$ are at the lower level. The pattern of communication is that first there is a call to the command c, then some number of repetitions of interaction pairs $\overline{c}\overline{r}$, then finally a return r.

The picture this gives of the assembly of a complete system is that one has a series of boxes, with input arrows linked to output arrows by a "twisted pair of wires" reminiscent of the Greek letter "χ". This is indeed a kind of composition in the categorical sense, where the morphisms are components. The paper is about this category.



How can we describe interfaces? Interface description languages (such as IDL from `http://www.omg.org/`) commonly take the form of signatures, *i.e.* typed procedure declarations. The type system is "simply typed", and it is used in connection with encoding and decoding arguments for possible remote transmission. It addresses other mechanistic, low-level and administrative issues. However an interface description ought to describe everything necessary to design and verify the correctness of a program that uses the interface, without knowing anything about how it might be implemented. It should state with complete precision a *contract*, or in Dijkstra's words a "logical firewall" between the user and implementer of an interface.

We define this category in (essentially) Martin-Löf's type theory, a constructive and predicative type theory in which the type-structure, is sufficiently rich to express specifications of interfaces with full precision. One reason for working in a constructive type theory is that a model for program components in such a setting is *ipso facto* a "working" model. In principle, one may write executable program components in this framework, and exploit type-checking to ensure that they behave correctly. In practice, one has to code programs in real programming languages. Nevertheless, one can perhaps develop programs in a dependently typed framework, using type-checking to guide and assist the development (as it were a mental prosthesis), run the programs to debug the specifications, and then code the programs in a real programming notation.

Our model is constructed from well-known ingredients. Since the seminal work of Floyd, Dijkstra and Hoare [12, 10, 20] there has been a well established tradition of specifying commands in programming languages through use of predicate transformers, and roughly speaking the objects of our category are predicate transformers on a state-space. Equally well established is the use of simulation relations to verify implementations of abstract data types, and roughly speaking, the morphisms of our category are simulation relations, or more precisely, relations together with a proof that they are simulations. The computational content of a simulation is contained in this (constructive) proof.

However, the "natural habitat" of the notions of predicate transformer and simulation is higher-order (impredicative) logic. To express these notions in a *predicative* framework, we work instead with concrete, first-order representations in which their computational content is made fully explicit. Again, the key ideas are fairly well-known, this time in the literature of constructive mathematics and predicative type theory. Our contribution is only to put them to use in connection with imperative programming.

Finally, our excuse for submitting a paper on programming to a conference on formal topology is that our category of interfaces and components turns out to coincide almost exactly with the category of basic topologies and basic continuous relations in Sambin's approach to formal topology. At the least, one can hope that further development of this approach to program development can benefit from research in the field of formal topology. One may also hope that work in formal topology can benefit in some way from several decades of intensive research in the foundations of imperative programming and perhaps even gain a new application area.

1.1 Plan of the paper

The first main section (2) begins with two ways in which the notion of subset can be expressed in type theory. Then set up some machinery for dealing with binary relations, to illustrate how our notions of subset have repercussions on higher order notions. In essence, we obtain besides the ordinary notion of relation a more computationally oriented notion of *transition structure*, that pre-figures our representation of predicate transformers.

The next two sections (3 and 4) concern the notion of monotone predicate transformer. In the first of these sections (3), we review the notion of predicate transformer as it occurs in the theory of inductive definitions and in the semantics of imperative programming. The main points here are that predicate transformers form a complete lattice under pointwise inclusion, that they possess also a monoidal structure of sequential composition, and moreover that there are two natural forms of "iteration". Section 4 is devoted to a predicative analysis of the notion of predicate transformer. This exploits the distinction drawn in section 2 between our two forms of the notion of subset. We represent predicate transformers by objects called *interaction structures,* and show that our representations supports the same algebraic structure.

The objects of our category are interaction structures over a set of states. The next section (section 5) is about morphisms between these objects. It is convenient to unfold our answer in three stages. In the first step we define a restricted notion of *linear simulation* (that is indeed connected with the linear implication of linear logic) for which an interaction in the domain is simulated by exactly one interaction in the codomain. In the second step, we move to the Kleisli category for a monad connected with the reflexive and transitive closure of an interaction structure; we call the morphisms in the Kleisli category *general simulations.* In the third and last step, taking a hint from formal topology, we take a quotient of general simulations, by passing to the *saturation* of a relation. The last step captures the idea that two relations may have the same simulating potential, modulo some hidden interactions.

Up to this point, the constructions have been motivated by considerations from imperative programming. In section 6, we examine the connection with formal topology. Firstly, our category of interaction structures and general morphisms corresponds exactly to Sambin's category of inductively defined basic topologies. Secondly, formal topology goes beyond basic topology by adding a notion of convergence, that allows for an analysis of the notion of point. The remainder of section 6 is concerned with a tentative interpretation of this extra structure.

We conclude with some questions raised in the course of the paper, and acknowledgment of some of the main sources of our ideas.

1.2 Mathematical framework(s)

We work in a number of different foundational settings, that we have tried to stratify in the following list.

- At the bottom, the most austere is Martin-Löf's type theory ([29, 32]), with a principle of inductive definitions similar to that used by Petersson and Synek in the paper [33], with certain forms of universe type, but without any form of propositional equality.

  Our category of interfaces and components can be defined using only predicative type theory with inductive definitions. In fact the category has been defined and its basic properties proved in such a theory using the

Agda “programming” language ([8]). The proof scripts can be found at http://iml.univ-mrs.fr/ hyvernat/academics.html.

- To this we add rules for propositional equality, which is necessary to round-out the programming environment to a language for fully constructive (intuitionistic and predicative) mathematics.

  This is not the right place to try to analyze the notion of equality, in any of its manifestations: definitional, propositional, judgmental, intensional, extensional and so on. It is however a source of non-computational phenomena in type theory, and the history of predicative type theory (if not also its future) is one of a constant struggle with this notion. We wish to carefully track the use of the equality relation (and cognate notions such as singleton predicate). That is we prefer to work with “pre-sets” rather than “setoids” [21].

- We also add a principle for coinductive definitions. The foundations of coinduction in predicative mathematics are not yet entirely clear. We simply use co-inductive definitions in the most “straightforward” way, meaning by this that our constructs seem to make good computational sense. One reference for the kind of coinductive definitions we will use can be found in [19].

- At various points, it seems necessary to relax the stricture of predicativity. In particular, we invoke the Knaster-Tarski theorem. This lacks a strictly predicative justification. Since we are trying to devise computationally-oriented analogues of certain impredicative constructions, it is necessary to look at matters from the impredicative point of view, if only for comparison.

- Finally, at the highest or most abstruse level, we shall occasionally make use of classical, impredicative reasoning, thus going beyond any straightforward computational interpretation. Working at this level Hyvernat ([23, 22]) has identified surprising connections between an impredicative variant of our category and classical linear logic, even of second order.

1.3 Type theoretic notation

Our notation is based (loosely) on Martin-Löf’s type theory, as expounded for example in [29, 32]. In the paper we call this simply “type theory”.

- To say that a value v is an element of a set S, we write $v \in S$. On the other hand, to say that o is an object of a proper type T (such as Set, the type of sets), we write $o : T$.

- We use standard notation as in, for example [29, 32], for indexed cartesian products and disjoint unions. This is summarized in the following table:

	product	sum
dependent version	$(\Pi\, a \in A)\, B(a)$	$(\Sigma\, a \in A)\, B(a)$
non-dependent	$A \to B$	$A \times B$
element in normal form	$(\lambda\, a \in A)\, b$	(a, b)

We iterate those constructions with a comma. Using the Curry-Howard isomorphism, we might also use the logical $\forall$ and $\exists$ as notations for Π and Σ.
We use the same notation at the type level.

- Instead of the binary disjoint union $A + B$, we prefer to use a notation in which constructors can be given mnemonic names, as is common in programming environments based on type theory. For example, the disjoint union $A + B$ itself could be written **data** $\mathsf{in}_0(a \in A) \mid \mathsf{in}_1(b \in B)$. As the eliminative counterpart of this construction, we use pattern matching.

  We also use ad-lib pattern matching in defining functions by recursion, rather than explicit elimination rules (recursors, or "weakly initial arrows").

- We use simultaneous inductive definitions of a family of sets over a fixed index-set (as in [33, 32]), with similar conventions.

  At an impredicative level, we will make use of μ-expressions for inductively defined sets, predicates, relations, and predicate transformers.

2 Two notions of subset

We will be concerned with two notions of subset, or more accurately two forms in which a subset of a set S may be given:

$$\{\, s \in S \mid U(s) \,\} \qquad \text{or} \qquad \{\, f(i) \mid i \in I \,\} \ .$$

The first we call "predicate form" —U is a predicate or propositional function with domain S. The second we call "indexed form", or "family form" —f is a function from the index set I into S. Other terminology might be "comprehension" versus "parametric", or "characteristic" versus "exhaustive".

For example, here are two ways to give the unit circle in the Euclidean plane: (note that we do not require in indexed form that the function f is injective)

$$\big\{\, (x,y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1 \,\big\} \qquad \text{or} \qquad \{\, (\sin\theta, \cos\theta) \mid \theta \in \mathbb{R} \,\} \ .$$

Of course, what we write in one form we may write in the other:

$$\begin{array}{ll} \{\, s \mid (s, \_) \in (\Sigma\, s \in S)\, U(s) \,\} \ ; & \textit{(predicate rewritten as family)} \\ \{\, s \in S \mid (\exists i \in I)\, s =_S f(i) \,\} \ . & \textit{(family rewritten as predicate)} \end{array}$$

To turn a predicate into an indexed family, we take as index the set of proofs that some elements satisfy the predicate, and for the indexing function the first projection. To turn an indexed family into a predicate, we make use of the equality relation "$=_S$" between elements of S, and in essence form the union of a family of singleton predicates: $\bigcup_{i \in I} \{f(i)\}$.

So it may seem that what we have here is a distinction without any real difference. Note however that the essence of a predicate is a (set-valued) function defined *on* S, while the essence of an indexed family is a function *into* S, so that there is a difference in variance. To make this clear, let us define two functors which take a set S to the type of predicate-form subsets of S, and to the type of indexed-form subsets of S.

Definition 1 *Define the following operations:*

- $Pow(S) \triangleq S \to Set$*, where we may write* $U : Pow(S)$ *as* $\{\, s \in S \mid U(s) \,\}$*; and if* $f \in S_1 \to S_2$*, then*
  $$\begin{array}{lcllcl} Pow(f) & : & Pow(S_2) & \to & Pow(S_1) \\ & & U & \mapsto & \big\{\, s_1 \in S_1 \mid U\big(f(s_1)\big) \,\big\} \end{array}$$
  We write $U \subseteq S$ *as a synonym for* $U : Pow(S)$*. Note that* $Pow(f)$ *is usually written* f^{-1}*.*

- $Fam(S) \triangleq (\Sigma\, I : Set)\, S \to I$*, where we may write* $(I, x) : Fam(S)$ *as* $\{\, x(i) \mid i \in I \,\}$*; and if* $f \in S_1 \to S_2$*, then*
  $$\begin{array}{lcllcl} Fam(f) & : & Fam(S_1) & \to & Fam(S_2) \\ & & \{\, x(i) \mid i \in I \,\} & \mapsto & \{\, f(x(i)) \mid i \in I \,\} \end{array}$$

The first functor is contravariant, while the second is covariant. So the distinction we have made corresponds after all to a well-known (even banal) difference.

In a predicative framework, both these functors cross a "size" boundary: they go from the category of (small) sets to the category of (proper) types. In fact these functors can be extended to endo-functors at the level of types, going from the category of (proper) types to itself. Remark however that the translations between subsets and families can not be carried out in either direction at the level of types.

- Going from families to subsets would require a propositional (*i.e.* set-valued) equality relation between the objects of arbitrary types, rather than merely between the elements of a set.

- Going from a propositional function defined on a type to an indexed family is in general impossible since we require the indexing set to be ... a set.

This will become important when we iterate or compose our two variants of the power-functor.

If we call into question, or try to work without the idea of a generic notion of propositional equality, the two notions of subset fall into sharp relief. In basic terms, the intuition of the distinction is that a family is something computational, connected with what we "do" or produce. On the other hand, a predicate is something specificational, connected with what we "say" or require.

How does the algebraic structure of predicates compare with that of indexed families? As for predicates, the situation is the normal one: if we interpret the logical constants constructively, they form a Heyting algebra. With the equality relation, the lattice is atomic, with singleton predicates for atoms. The inclusion and "overlap" relations are defined as follows:

Definition 2 *Let* U *and* V *be two subsets of the same set* S*; define:*

- $s \,\epsilon\, U \triangleq U(s)$ *(i.e.* $s \,\epsilon\, U$ *iff "*$U(s)$ *is inhabited");*

- $U \subseteq V \triangleq (\Pi\, s \in S)\, U(s) \to V(s)$*;* *(i.e.* $(\forall s \in S)\ s \,\epsilon\, U \to s \,\epsilon\, V$*)*

- $U \between V \triangleq (\Sigma\, s \in S)\, U(s) \wedge V(s)$*.*

The importance of $\between$ in a constructive setting has been stressed by Sambin: it is a positive version of non-disjointness, dual to inclusion.

Remark. The confusion between the two meanings of "$\subseteq$" can always be resolved ("$\subseteq$" is a synonym for $\_ : \mathit{Pow}(\_)$ and denotes inclusion of subsets). For a full account of traditional set theoretic notions in "subset theory", we refer to [39]. Here are two examples:

- $S_{\mathrm{Full}} \triangleq \{\, s \in S \mid \top \,\}$ contains all the elements of S. We write it simply S;
- $U \times V \triangleq \{\, (s, s') \in S \times S \mid s \mathrel{\epsilon} U \text{ and } s' \mathrel{\epsilon} V \,\}$.

What now about families? In the presence of equality, which allows us to pass from a family to the corresponding predicate, their algebraic structure is the same as that of predicates. However, if we abstain from use of equality, the situation is as follows. The construction of set indexed suprema can be carried through

$$\textstyle\bigcup_{i \in I} \{\, f_i(t) \mid t \in T_i \,\} \quad \triangleq \quad \{\, f_i(t) \mid (i,t) \in (\Sigma\, i \in I)\, T_i \,\} \ ,$$

which gives a sup-lattice. Additionally for any $s \in S$ we can form the singleton family $\{\, s \mid i \in I \,\}$ taking for I any non-empty set.

We cannot *say* that an element of S belongs to a family $\{\, f(i) \mid i \in I \,\}$. Still less can we say that one family includes another, or overlaps with it (as this requires an equation). What we *can* state however is that a family is included in a predicate, or that it overlaps with it:

$$\begin{array}{rcl} \{\, f(i) \mid i \in I \,\} \subseteq U & \triangleq & (\forall i \in I)\, U\big(f(i)\big) \ ; \\ \{\, f(i) \mid i \in I \,\} \between U & \triangleq & (\exists i \in I)\, U\big(f(i)\big) \ . \end{array}$$

To summarize, predicates have a rich algebraic structure. In contrast, the structure of families is impoverished, supporting only suprema operations of various kinds. To compensate, we have a concrete, computational form of the notion of subset.

2.1 The general notion of binary relation

A binary relation between two sets S_1 and S_2 is a subset of the cartesian product $S_1 \times S_2$, or to put it another way, a function from S_1 to subsets of S_2:

$$\begin{array}{rcl} \mathit{Pow}(S_1 \times S_2) & = & (S_1 \times S_2) \rightarrow \mathit{Set} \\ & \simeq & S_1 \rightarrow (S_2 \rightarrow \mathit{Set}) \\ & = & S_1 \rightarrow \mathit{Pow}(S_2) \ . \end{array}$$

We will leave implicit the isomorphism ("currying") between the two versions. There are thus two ways to write "s_1 and s_2 are related through $R \subseteq S_1 \times S_2$": either "$(s_1, s_2) \mathrel{\epsilon} R$" or "$s_2 \mathrel{\epsilon} R(s_1)$".

Because relations are subset valued functions, they inherit all the algebraic structure of predicates pointwise. Additionally, we can define the following operations.

Converse: $\dfrac{R \subseteq S_1 \times S_2}{R^\sim \subseteq S_2 \times S_1}$ with $(s_2, s_1) \mathrel{\epsilon} R^\sim \triangleq (s_1, s_2) \mathrel{\epsilon} R$.

Equality: $\mathsf{eq} \subseteq S \times S$ with $\mathsf{eq}(s) = \{s\}$. *(This requires equality!)*

Composition: $\dfrac{Q \subseteq S_1 \times S_2 \qquad R \subseteq S_2 \times S_3}{Q \fatsemi R \subseteq S_1 \times S_3}$

with $(s_1, s_3) \mathrel{\epsilon} (Q \fatsemi R) \triangleq (\exists s_2 \in S_2)\ (s_1, s_2) \mathrel{\epsilon} Q \text{ and } (s_2, s_3) \mathrel{\epsilon} R$.

Reflexive and transitive closure: $\dfrac{R \subseteq S \times S}{R^* \subseteq S \times S}$

with $R^* \triangleq \mathsf{eq} \cup R \cup (R \fatsemi R) \cup R^3 \cup \ldots$ *(inductive definition)*

Note that the "reflexive" part requires equality to be definable.

Post and pre-division:

- $\dfrac{Q \subseteq S_1 \times S_3 \qquad R \subseteq S_2 \times S_3}{(Q \mathbin{/} R) \subseteq S_1 \times S_2}$

  with $(s_1, s_2) \mathrel{\epsilon} (Q \mathbin{/} R) \triangleq R(s_2) \subseteq Q(s_1)$;

- $\dfrac{Q \subseteq S_1 \times S_3 \qquad R \subseteq S_1 \times S_2}{(R \setminus Q) \subseteq S_2 \times S_3}$ with $(R \setminus Q) \triangleq (Q^{\sim} / R^{\sim})^{\sim}$.

These operators satisfy a wealth of familiar algebraic laws, from which we want to recall only the following.

- Composition and equality are the operators of a monoid. Composition is monotone in both arguments, and in fact commutes with arbitrary unions on both sides.

- Post-composition $(\_ \fatsemi R)$ is left-adjoint to post-division $(\_ \mathbin{/} R)$; similarly, pre-composition $(R \fatsemi \_)$ is left-adjoint to pre-division $(R \setminus \_)$.

- Converse is involutive and reverses composition: $(Q \fatsemi R)^{\sim} = R^{\sim} \fatsemi Q^{\sim}$.

- For each function $f \in S_1 \rightarrow S_2$, its graph relation $\mathbf{gr}\, f \subseteq S_1 \times S_2$ satisfies both $\mathsf{eq}_{S_1} \subseteq (\mathbf{gr}\, f) \fatsemi (\mathbf{gr}\, f)^{\sim}$ (totality), and $(\mathbf{gr}\, f)^{\sim} \fatsemi \mathbf{gr}\, f \subseteq \mathsf{eq}_{S_2}$ (determinacy).

2.2 Transition structures

What happens to the notion of a binary relation if we replace the contravariant functor $\mathit{Pow}(\_)$ with the co-variant functor $\mathit{Fam}(\_)$? This gives two candidates for a computational representation of relations:

$$\mathit{Fam}(S_1 \times S_2) \qquad \text{and} \qquad S_1 \rightarrow \mathit{Fam}(S_2) \ .$$

- In more detail, an object of the first type consists of a set I, together with a pair of functions with I as their common domain: $f \in I \rightarrow S_1$ and $g \in I \rightarrow S_2$. Such a pair is commonly known as a *span*.

- On the other hand, an object T of the second type consists of a function F which assigns to each $s \in S_1$ a family of S_2's, that we may write

$$F(s) = \{\, n(s,t) \mid t \in A(s) \,\} \ .$$

Where $A : S_1 \to \mathit{Set}$ and $n \in (\Pi\, s_1 \in S_1)\, A(s_1) \to S_2$. We call this a *transition structure*. When no confusion arises, we write "$s[a]$" instead of "$n(s,a)$".

In contrast with the situation with relations, any isomorphism that can be defined between spans and transition structures seems to require use of an equality relation. Transition structures are inherently asymmetric. There is a genuine bifurcation between spans and transition structures. In this paper we shall be concerned only with transition structures. To some extent, the relationship between spans and transition structures remains to be explored.

Transition structures sometimes provide a more appropriate model than relations for "asymmetric" situations in which one of the terms of the relation has priority or precedence in some sense.

- The notion of an occurrence of a subexpression of a first-order expression can be represented by a transition structure on expressions, in which the set $A(s)$ represents the set of positions within s, and $s[a]$ represents the subexpression of s that occurs at position a.
- In general rewriting systems, an expression is rewritten according to a given set of rewriting rules. In state s, each rule can be represented by an $a \in A(s)$, where $s[a]$ is the result of the rewriting of s by the rule a.
- A deterministic automaton that reads a stream of characters, changing state in response to successive characters can be represented by a transition structure. In such a case, one usually writes $s \xrightarrow{a} s'$ for $s[a] = s'$.

In comparison with relations, transition structures have weaker algebraic properties. There are transition structure representations for equality relations and more generally the graphs of functions, and for indexed unions, composition, and closure operations such as reflexive and transitive closure: transition structures form a Kleene algebra.

Composition:
$$\frac{T_1 : S_1 \to \mathit{Fam}(S_2) \qquad T_2 : S_2 \to \mathit{Fam}(S_3)}{(T_1 \fatsemi T_2) : S_1 \to \mathit{Fam}(S_3)}$$

where the components $(T_1 \fatsemi T_2).A$ and $(T_1 \fatsemi T_2).n$ of $T_1 \fatsemi T_2$ are defined as:

$$\begin{array}{lcl} (T_1 \fatsemi T_2).A(s_1) & \triangleq & (\Sigma\, t_1 : T_1.A(s_1))\, T_2.A(s_1[t_1]) \\ (T_1 \fatsemi T_2).n(t_1, t_2) & \triangleq & (s_1[t_1])[t_2] \ . \end{array}$$

Identity: $\mathtt{eq} : S \to \mathit{Fam}(S)$ with $T(\_) \triangleq \{*\}$ and $s[\_] \triangleq s$. Note that the equality relation is not necessary to define this interaction structure.

The definitions are straightforward, and the reader is encouraged to try the case of reflexive and transitive closure for themselves.

On the other hand, transition structures are not closed under intersection, converse, or division. They can however be used as pre-components to relations, and as post-divisors of relations. The definitions, which make no use of equality, are as follows.

$$\begin{array}{lcl} (s_1, s_3) \mathrel{\epsilon} (T \fatsemi R) & \triangleq & T(s_1) \between R^{\sim}(s_3) \ ; \\ (s_1, s_2) \mathrel{\epsilon} (R \,/\, T) & \triangleq & T(s_2) \subseteq R(s_1) \ . \end{array}$$

(In the first equation, $T : S_1 \to \mathit{Fam}(S_2)$ and $R \subseteq S_2 \times S_3$, while in the second, $R \subseteq S_1 \times S_3$ and $T : S_2 \to \mathit{Fam}(S_3)$.)

Note that we can define the relation corresponding to a transition structure by precomposing the transition structure to equality: if $T : S_1 \to \mathit{Fam}(S_2)$, define $T^\circ : S_1 \to \mathit{Pow}(S_2)$ as $T \mathbin{⨾} \mathsf{eq}_{S_2}$.

3 Predicate transformers

3.1 Motivations and basic definitions

A predicate transformer is a function from subsets of one set to subsets of another:

$$\begin{array}{rcl} \mathit{Pow}(S_2) \to \mathit{Pow}(S_1) & = & \mathit{Pow}(S_2) \to S_1 \to \mathit{Set} \\ & \simeq & \big(\mathit{Pow}(S_2) \times S_1\big) \to \mathit{Set} \\ & \simeq & \big(S_1 \times \mathit{Pow}(S_2)\big) \to \mathit{Set} \\ & \simeq & S_1 \to \big(\mathit{Pow}(S_2) \to \mathit{Set}\big) \\ & = & S_1 \to \mathit{Pow}\,\big(\mathit{Pow}(S_2)\big)\ . \end{array}$$

As these isomorphisms show, from another point of view, a predicate transformer is nothing but a higher-order relation (between elements of one set and subsets of another).

Since the mid-70's, predicate transformers have been used as denotations for commands such as assignment statements in imperative programming languages. Some predicate transformers commonly considered in computer science are the weakest precondition operator, the weakest liberal precondition, the strongest postcondition (all introduced by Dijkstra), and the weakest and strongest invariant of a concurrent program (introduced by Lamport). Perhaps the most fundamental of these is the weakest precondition. In weakest precondition semantics, one associates to a program statement P a predicate transformer $|P|$ mapping a *goal* predicate (which one would like to bring about) to an *initial* predicate (which ensures that execution of P terminates in a state satisfying the goal predicate). On the other hand, the weakest liberal precondition is more relevant in connection with predicates which one would like to avoid or maintain.

In an effort to cut down the semantic domain of predicate transformers to those that are in some sense executable, various "healthiness" properties[1] have been required of predicate transformers. In the 80's and 90's reasons emerged for relaxing most such restrictions, except for the most basic, monotonicity. In explanation of monotonicity, if a goal predicate is weakened (made easier to achieve), the corresponding initial predicate should be weakened. More technically, the Knaster-Tarski theorem is heavily exploited in developing the semantics of recursion and iteration. In the following, the qualification "monotone" will be implicit: all predicate transformers will be monotone, except where explicitly indicated.

An active field of computer science instigated by Morgan, Morris, Back, and von Wright is now founded on the use of monotone predicate transformers

[1] like strictness, distribution over intersections, distribution over directed unions

not just as a semantic domain for commands, but as a framework for developing imperative programs from specifications. This field is called "refinement calculus"; the canonical reference for the refinement calculus is Back and von Wright's textbook [4].

The refinement calculus is a "wide spectrum" language in the sense that both programs and specifications are represented by monotone predicate transformers. (In contrast, in type theory programs and specifications lie, roughly speaking, on opposite sides of the "$\in$" symbol.) Specifications are manipulated into an executable form (acquiring various healthiness conditions), until they can be coded in a real programming notation.

3.2 Algebraic structure

The lattice structure of predicates lifts pointwise to the level of relations. Analogously, the lattice structure lifts to the level of predicate transformers:

- predicate transformers are ordered by pointwise inclusion:

$$F \subseteq G \quad \triangleq \quad \text{“}(\forall U \subseteq S)\ F(U) \subseteq G(U)\text{”} \ ;$$

  i.e. "$F \subseteq G$" is a shorthand for the *judgment* "$U \subseteq S \vdash F(U) \subseteq G(U)$" and is not an actual proposition or set.

- they are closed under intersection and union:

$$\begin{array}{rcl} \big(\bigcup_i F_i\big)(U) & \triangleq & \bigcup_i \big(F_i(U)\big) \ ; \\ \big(\bigcap_i F_i\big)(U) & \triangleq & \bigcap_i \big(F_i(U)\big) \ . \end{array}$$

The bottom and top of the lattice are conventionally called abort and magic respectively. The predicate transformer abort transforms all predicates to the empty predicate: it is impossible to achieve anything by use of a resource satisfying abort. On the other hand, magic transforms all predicates to the trivial predicate, which always holds. A resource fulfilling the magic specification could be used to accomplish anything, even the impossible.

Just as relations support not only a lattice structure, but also a monoidal structure of composition, so it is with predicate transformers. Predicate transformers are of course closed under composition:

$$F \mathbin{⨾} G \triangleq F \cdot G \ ;$$

and the unit of composition is conventionally called skip:

$$\mathsf{skip}(U) \triangleq U \ .$$

Both relational and predicate transformer compositions are monotone. The distributivity laws satisfied by "⨾" are however quite different from the case of relations. With relations, composition distributes over unions on both sides, though not (in general) over intersections. With predicate transformers, composition distributes over both intersections and unions on the left, though not in general over either intersection or union on the right.

3.3 Angelic and demonic update

Somewhat as a function $f \in S_1 \to S_2$ lifts to a relation $(\mathbf{gr}\, f) : S_1 \to \mathit{Pow}(S_2)$, so a relation $R : S_1 \to \mathit{Pow}(S_2)$ lifts to a predicate transformer. However in this case there are two lift operations. These are conventionally called angelic and demonic update.

$$\frac{R : S_1 \to \mathit{Pow}(S_2)}{\langle R\rangle, [R] : \mathit{Pow}(S_2) \to \mathit{Pow}(S_1)}$$

with:[2]

$$\begin{array}{lcll} \langle R\rangle(U) & \triangleq & \{\, s_1 \in S_1 \mid R(s_1) \between U \,\} \ ; & \textit{(angelic update)} \\ {[R]}(U) & \triangleq & \{\, s_1 \in S_1 \mid R(s_1) \subseteq U \,\} \ . & \textit{(demonic update)} \end{array}$$

Note also that $\langle R^{\sim}\rangle(U)$ is nothing but the set of states related by $R^{\sim}$ to states that satisfy U or, in other words, the direct relational image of U under R. When there is no danger of confusion, we shall in the following write $R(U)$ for $\langle R^{\sim}\rangle(U)$ and R for $\langle R^{\sim}\rangle$.

At first sight, the angelic and demonic updates may look a little strange. What do they have to do with programming? In two particular cases though, they are immediately recognizable, namely when firstly, the relation is included in the equality relation on a state-space; and secondly when the relation is the graph of a function.

Assertions and assumptions: when the relation R is a subset of the identity relation (which can be identified with a predicate U), the angelic update $\langle U\rangle$ is known as an *assertion* (that the Angel is obliged to prove), whereas the demonic update $[U]$ is known as an *assumption* (that the Demon is obliged to prove). Assertion and assumption satisfy the equivalences:

$$\langle U\rangle(V) \ = \ U \cap V \qquad \text{and} \qquad [U](V) \ = \ \{\, s \in S \mid U(s) \to V(s) \,\} \ .$$

Assignments: because singleton predicates $\{s\}$ satisfy the equivalences

$$\{s\} \between U \quad \Leftrightarrow \quad s \mathrel{\epsilon} U \quad \Leftrightarrow \quad \{s\} \subseteq U \ ,$$

it follows that if $f \in S_1 \to S_2$, we have $\langle \mathbf{gr}\, f\rangle(U) \simeq U \cdot f \simeq [\mathbf{gr}\, f](U)$. In this case the predicate transformer commutes with arbitrary intersections and unions. The canonical example of such an update is the assignment statement $x := e$ where x is a state variable, and e is a "side-effect free" mathematical expression that may refer to the values of other state variables. This is interpreted as the "substitution" predicate transformer $U \mapsto \{\, s \in S \mid f(s) \mathrel{\epsilon} U \,\}$, where $f \in S \to S$ is the function that maps a state s to the state s' in which all variables except x have the same value as in s, and the value of x in s' is the denotation of the expression e in state s.[3]

[2] Note that we have diverged slightly from the notation of Back and von Wright. In their notation, the angelic update $\langle R\rangle$ is written $\{R\}$.

[3] It would take us too far afield to fully explain the syntax and semantics of state variables and assignment statements.

3.4 Fundamental adjunction

Perhaps the most fundamental law in the refinement calculus, with the same pivotal rôle as Sambin's "fundamental adjunction" in his development of basic topology through basic pairs ([35]) is the following Galois connection between angelic and demonic updates.

Proposition 1 *Suppose $R \subseteq S_1 \times S_2$; we have, for all $U \subseteq S_1$, $V \subseteq S_2$*

$$\langle R^{\sim}\rangle(U) \subseteq V \quad \Leftrightarrow \quad U \subseteq [R](V) \ ,$$

which is commonly written $\langle R^{\sim}\rangle \dashv [R]$.

Proof: Straightforward. □

Points *1* and *2* of the following corollary are the ground for all the development of basic topology from "basic pairs" ([35]). Recall that an interior [closure] operator is a predicate transformer P satisfying:

closure	*interior*
$U \subseteq P(U)$	$P(U) \subseteq U$
$U \subseteq P(V) \Rightarrow P(U) \subseteq P(V)$	$P(U) \subseteq V \Rightarrow P(U) \subseteq P(V)$

Corollary 1 *We have:*

1. *$\langle R^{\sim}\rangle \mathbin{;} [R]$ is an interior operator, in particular: $\langle R^{\sim}\rangle \mathbin{;} [R] \subseteq \mathsf{skip}$;*
2. *$[R] \mathbin{;} \langle R^{\sim}\rangle$ is a closure operator, in particular: $\mathsf{skip} \subseteq [R] \mathbin{;} \langle R^{\sim}\rangle$;*
3. *$[R] = [R] \mathbin{;} \langle R^{\sim}\rangle \mathbin{;} [R]$ and $\langle R^{\sim}\rangle = \langle R^{\sim}\rangle \mathbin{;} [R] \mathbin{;} \langle R^{\sim}\rangle$;*
4. *$\langle R^{\sim}\rangle$ commutes with all unions and $[R]$ commutes with all intersections.*

Proof: Straightforward. □

Back and von Wright's textbook on the refinement calculus contains many normal form theorems that relate the properties of a predicate transformer to its expression in the refinement calculus. Among these, the most general is the following. It provides one motivation for the analysis of predicate transformers given in section 4 below.

> **Theorem 13.10.** Let S be an arbitrary monotonic predicate transformer term. Then there exist state relation terms P and Q such that $S = \langle P\rangle \mathbin{;} [Q]$. ([4], p. 220, with $\{Q\}$ changed to $\langle Q\rangle$)[4]

In other words, so far as monotone predicate transformers are concerned, it suffices to consider those in which an angelic update is followed by a demonic update. In section 4, we will represent predicate transformers by such a composition, where the update relations are each given by transition structures.

[4] The proof given is an manipulation in higher-order logic, in which the relation Q is taken to be the membership relation.

3.5 Iterative constructions

The most interesting construct are connected with iteration. (One of the main applications of our category will be to model iterative client-server interaction, in section 4.5.)

In the case of relations and transition structures, there is a single notion of iteration, namely the reflexive and transitive closure. However in the case of predicate transformers, there are two different iteration operators: one orientated toward the Angel, and the other toward the Demon.

According to the Knaster-Tarski theorem, each monotone predicate transformer $F : Pow(S) \to Pow(S)$ possesses both a least fixpoint μF and a greatest fixpoint νF. They can be defined as:

$$\begin{array}{lcl} (\mu\, X)\, F(X) & \triangleq & \bigcap\{\, U \subseteq S \mid F(U) \subseteq U \,\} : Pow(S)\ ; \\ (\nu\, X)\, F(X) & \triangleq & \bigcup\{\, U \subseteq S \mid U \subseteq F(U) \,\} : Pow(S)\ . \end{array}$$

Note that the intersection and union operators are applied to a higher order predicate (a predicate of predicates, rather than a family of predicates). In a predicative framework we therefore run into difficult questions about the justification of those very general forms of induction and coinduction. In this paper we attempt no answer to these foundational questions: we need to consider only certain forms of "tail" recursion, in which the μ- or ν-bound variable occurs only as the right-hand operand of $\fatsemi$.

The two operations we need are written $\_^*$ and $\_^\infty$, and are characterized by the laws:[5]

$$\frac{F : Pow(S) \to Pow(S)}{F^*, F^\infty : Pow(S) \to Pow(S)}$$

with the rules:

$$\begin{array}{lcll} \mathsf{skip} \cup (F \fatsemi F^*) \subseteq F^* & , & \dfrac{\mathsf{skip} \cup (F \fatsemi G) \subseteq G}{F^* \subseteq G} & ; \\[2ex] F^\infty \subseteq \mathsf{skip} \cap (F \fatsemi F^\infty) & , & \dfrac{G \subseteq \mathsf{skip} \cap (F \fatsemi G)}{G \subseteq F^\infty} & . \end{array}$$

We may define these operations using μ and ν as:

$$F^*(U) \ \triangleq\ (\mu\, X)\, U \cup (F \fatsemi X) \qquad \text{and} \qquad F^\infty(V) \ \triangleq\ (\nu\, X)\, V \cap (F \fatsemi X)\ .$$

Both are iterative constructions. In the case of F^* the iteration must be finite and the Angel chooses when to exit. In the case of F^∞, the iteration may be infinite, and the Demon chooses when (if ever) to exit.

Proposition 2 *If F is a predicate transformer, then F^* is a closure operator and F^∞ is an interior operator.*

[5] Yet again we diverge from (and indeed clash with) the usual notation of Back and von Wright's refinement calculus. What we call angelic iteration, and write F^* is written there $F^{\emptyset}$ (and also called angelic iteration). What we call demonic iteration and write F^∞ is written there F^*, and called weak iteration.

Proof: We give only the proof that F^* is a closure operator. The proof that F^∞ is an interior operator is completely dual.

- $U \subseteq F^*(U)$: we know that $F^*(U)$ is a pre-fixpoint of $X \mapsto U \cup F(X)$, which means that $U \cup F\big(F^*(U)\big) \subseteq F^*(U)$, and so $U \subseteq F^*(U)$.
- $U \subseteq F^*(V) \Rightarrow F^*(U) \subseteq F^*(V)$. Suppose that $U \subseteq F^*(V)$. Since $F^*(U)$ is the least pre-fixpoint of $X \mapsto U \cup F(X)$, it suffices to show that $F^*(V)$ is also a pre-fixpoint of this operator, *i.e.* that $U \cup F\big(F^*(V)\big) \subseteq F^*(V)$. Since $F^*(V)$ is a pre-fixpoint for $X \mapsto V \cup F(X)$, we have $F\big(F^*(V)\big) \subseteq F^*(V)$, and by hypothesis, we have $U \subseteq F^*(V)$. We can conclude.

It is worth noting that the operation * itself is a closure operation on the lattice of predicate transformers, but that $^\infty$ is *not* an interior operator.

□

Some other properties of those operations are given by the following lemma. First, a definition:

Definition 3 *Suppose F is a predicate transformer.*

1. *an F-*invariant*, (or simply an* invariant *when F is clear) is a post-fixpoint of F, i.e. a predicate U satisfying $U \subseteq F(U)$;*
2. *an F-*saturated *predicate, (or simply a* saturated *predicate when F is clear) is a pre-fixpoint of F, i.e. a predicate U satisfying $F(U) \subseteq U$.*

We have:

Lemma 3.1 *If F is a predicate transformer on S and $U \subseteq S$, we have:*

- *$F^*(U)$ is the strongest (i.e. least) F-saturated predicate including U;*
- *$F^\infty(U)$ is the weakest (i.e. greatest) F-invariant contained in U.*

Proof: We will prove only the second point, as the first one is completely dual.

- $F^\infty(U)$ is contained in U: this is a consequence of F^∞ being an interior operator. (Proposition 2.)
- $F^\infty(U)$ is F-invariant: $F^\infty(U)$ is the greatest post-fixpoint of the operator $X \mapsto U \cap F(X)$; in particular, $F^\infty(U) \subseteq U \cap F\big(F^\infty(U)\big)$, which implies that $F^\infty(U) \subseteq F\big(F^\infty(U)\big)$.
- $F^\infty(U)$ is the greatest such invariant: suppose that V is another invariant contained in U, *i.e.* we have $V \subseteq F(V)$ and $V \subseteq U$. This implies that V is a post-fixpoint of the above operator. Since $F^\infty(U)$ is the greatest post-fixpoint, we conclude directly that $V \subseteq F^\infty(U)$.

□

4 Interaction structures

4.1 Motivations

As in the case of relations, we obtain another more computationally oriented notion of predicate transformer by replacing the $Pow(\_)$ functor with the $Fam(\_)$

functor. There is again more than one way to do this. We will focus on the structure arising from the representation of predicate transformers as $S \to \mathit{Pow}^2(S')$:

$$w : S \to \mathit{Fam}^2(S') \ .$$

Expanding the definition of $\mathit{Fam}(\_)$, we see that the declaration of w consists of the following data:

1. a function $A : S \to \mathit{Set}$;
2. a function $D : (\Pi\, s \in S)\ \ A(s) \to \mathit{Set}$;
3. a function $n : (\Pi\, s \in S, a \in A(s))\ \ D(s,a) \to S'$.

In essentials, the invention of this structure should be attributed to Petersson and Synek (though similar constructions were implicitly present in earlier works: [41, 16, 11]). In [33], they introduced a set-constructor for a certain inductively defined family of trees, relative to the signature

$$\begin{array}{lll} A : \mathit{Set} & & \\ B(x) : \mathit{Set} & \text{where} & x \in A \\ C(x,y) : \mathit{Set} & \text{where} & x \in A, y \in B(x) \\ d(x,y,z) \in A & \text{where} & x \in A, y \in B(x), z \in C(x,y) \end{array}$$

which is nothing more than a pair of a set A and an element of $A \to \mathit{Fam}^2(A)$. We will make use of (a slight variant of) their definitional schema in defining one of our iteration operators below, namely "angelic iteration".

4.2 Applications of interaction structures

This type is rich in applications. Broadly speaking these applications fall under two headings: interaction and inference.

Interaction. This is our main application.

S: we take S to model the state space of a device. We prefer to call this the state of the *interface* as the device itself may have a complicated internal state which we need not understand to make use of the device. For example, think of $s \in S$ as the state of one's bank balance, as it is observed by someone using an ATM.[6]

A: for each state $s \in S$, we take the set $A(s)$ to model the set of commands that the user may issue to the device. For example, think of $a \in A(s)$ as a request to withdraw cash from an ATM.

D: For each $s \in S$ and $a \in A(s)$, we take $D(s,a)$ to model the set of responses that the device may return to the command a. It is possible that there is more than one response that the device may legitimately return to a given command. For example, think of the response `Service Unavailable` to a withdrawal request.

[6] Automatic Teller Machine —a cash machine.

n: For each $s \in S$, command $a \in A(s)$ and response $d \in D(s, a)$, we take $n(s, a, d)$ to model the next state of the interface. Note that the next state is *determined* by the response. This means that the current state of the system can always be computed from its initial state, together with a complete record of commands and responses exchanged *ab initio*.

The two agents interacting across such a command-response interface are conventionally called the Angel (for a pronoun we use "she"), and the Demon ("he"). The Angel issues commands and receives responses. She is active, in that she has the initiative in any interaction. The Demon is passive, and merely obeys instructions, to each of which he returns a response. The terminology of Angels and Demons is rife in the refinement calculus literature, in which an interface is thought of as a contract regulating the behavior of two parties, the Angel and Demon. We have named the two components of an interaction structure A and D after them. (Alternative *dramatis personae* might be ∃loise and ∀belard, Opponent and Defendant, Master and Slave, Client and Server.)

Other applications that have broadly the same interactive character are indicated in the following table.

idiom	S	A	D	n
game	state	moves	counter-moves	next state
system	state	system call	return	next state
experiment	knowledge	stimulus	response	
examination	knowledge	question	answer	

Inference. A second style of application of the structure (which plays no explicit rôle in this paper) is to model an inference system, or (to use Aczel's term) a rule-set. One does not attempt here to capture the idea of a schematic rule, but rather the inference steps that are instances of such rules.

S: we may take the elements of the set S to model judgments that can stand 'positively' at the conclusion or occur 'negatively' as some premise of an inference step.

A: for each judgment $s \in S$, we may take the elements of the set $A(s)$ to model inference steps with conclusion s.

D: for each judgment $s \in S$ and inference step $a \in A(s)$ by which s can be inferred, we may take the elements of the set $D(s, a)$ to index, locate, or identify one of the family of premises required to infer s by inference step a.

n: for each judgment $s \in A$, inference step $a \in A(s)$, and index $d \in D(s, a)$ for a premise of that inference step, we may take $n(s, a, d)$ to model the judgment to be proved at premise d in inference step a.

Instead of judgements and inference steps, we may consider grammatical categories and productions as in Petersson and Synek's original application ([33]), or sorts and (multi-sorted) signatures.

4.3 Definition and basic properties

Definition 4 *If S and S' are sets, an object w of type $S \rightarrow \mathit{Fam}^2(S')$ is called an* interaction structure *(from S to S'). We refer to the components of w as follows:*

$$\begin{array}{lcl} w.A & : & S \rightarrow \mathit{Set} \\ w.D & : & (\Pi\, s \in S)\ \ w.A(s) \rightarrow \mathit{Set} \\ w.n & \in & (\Pi\, s \in S,\ a \in w.A(s))\ \ w.D(s,a) \rightarrow S' \end{array}$$

When no confusion is possible, we prefer to leave the "w." implicit, and simply write A, D and n, possibly with decorations. We also use the notation $s[a/d]$ as a synonym for $w.n(a,s,d)$ when w is clear from the context.

Before examining the objects of this type in more detail, we mention some other representations of predicate transformers:

- $S_1 \rightarrow \mathit{Pow}^2(S_2) \simeq \mathit{Pow}(S_2) \rightarrow \mathit{Pow}(S_1)$: this is the notion studied in section 3, or in Back and von Wright's book ([4, sec. 5.1, p. 251]) under the name "choice semantics";

- $S_1 \rightarrow \mathit{Pow}\big(\mathit{Fam}(S_2)\big) \simeq \mathit{Fam}(S_2) \rightarrow \mathit{Pow}(S_1)$: this notion is very similar to the previous one (they are equivalent in the presence of equality). To our knowledge, this type has never been considered as a viable notion;

- $S_1 \rightarrow \mathit{Fam}\big(\mathit{Pow}(S_2)\big)$: because a subset on a proper type need not be equivalent to a set indexed family on the same type, this notion is intrinsically different from the previous two. This is the notion used by Aczel to model generalized inductive definitions in [2]. This is also the structure used in [9] under the name *axiom set*.

  From our perspective, this notion seems to abstract away the action of the Demon: the Angel doesn't see the Demon's reaction, but only a property of the state it produces. The Demon's reaction is in some sense "hidden".

There are other variants based on types isomorphic to $\mathit{Pow}(S_2) \rightarrow \mathit{Pow}(S_1)$ such as $\mathit{Fam}(S_2) \rightarrow \mathit{Fam}(S_1)$, $\mathit{Pow}(S_2 \times \mathit{Fam}(S_1))$ and so on. We have not investigated all the possibilities systematically, but none of them seems to fit our purpose.

Associated with an interaction structure w from S to S' are two monotone predicate transformers w° and $w^\bullet : \mathit{Pow}(S') \rightarrow \mathit{Pow}(S)$. Both are concerned with the notion of "reachability" of a predicate (on S') from a state (in S). The difference is which agent tries to bring about the predicate: either the Angel (in the case of w°) or the Demon (in the case of $w^\bullet$).

Definition 5 *If $w = (A, D, n)$ is an interaction structure on S, define:*
(recall that "$\exists$" and "$\forall$" are synonyms for "Σ" and "Π")

$$\begin{array}{lcl} s \mathrel{\epsilon} w^\circ(U) & \Leftrightarrow & \big(\exists a \in A(s)\big)\big(\forall d \in D(s,a)\big)\ s[a/d] \mathrel{\epsilon} U\ ; \\ s \mathrel{\epsilon} w^\bullet(U) & \Leftrightarrow & \big(\forall a \in A(s)\big)\big(\exists d \in D(s,a)\big)\ s[a/d] \mathrel{\epsilon} U\ . \end{array}$$

Of these, lemma 4.1 below shows that $\_^\circ$ is more fundamental.

Definition 6 *If S and S' are sets, and w is an interaction structure from S to S', define $w^\perp : S \to Fam^2(S')$ as follows.*

$$\begin{array}{lcl} w^\perp.A(s) & \triangleq & (\Pi\, a \in w.A(s))\;\; w.D(s,a) \\ w^\perp.D(s,\_) & \triangleq & w.A(s) \\ w^\perp.n(s,f,a) & \triangleq & s[a/f(a)] \end{array}$$

As we'll see in proposition 3, this is a constructive version of the dual operator on predicate transformers. Although this operation doesn't enjoy all the duality properties of its classical version (in particular, it is not provably involutive), we still have the following:

Lemma 4.1 *For any interaction structure w, we have: $w^\bullet = (w^\perp)^\circ$.*

Proof: Axiom of choice. □
The converse $w^\circ = (w^\perp)^\bullet$ holds classically but not constructively.

4.3.1 Lattice structure, monoidal operations

We define inclusion between interaction structures by interpreting them as predicate transformers via the $^\circ$ operator:

Definition 7 *Define $w_1 \subseteq w_2 \triangleq {w_1}^\circ \subseteq {w_2}^\circ$.*

Once again, this is not a proposition, but only a judgment.

In contrast with the impoverished structure of transition structures relative to relations, interaction structures support the full algebraic structure of monotone predicate transformers, as we now show.

Definition 8 *Define the following operations on interaction structures:*

Updates *If $T = (A, n) : S \to Fam(S')$ is a transition structure, then*

$$\begin{array}{lcl@{\qquad\qquad}lcl} \langle T\rangle.A(s) & \triangleq & A(s) & [T].A(s) & \triangleq & \{*\} \\ \langle T\rangle.D(s,a) & \triangleq & \{*\} & [T].D(s,\_) & \triangleq & A(s) \\ \langle T\rangle.n(s,a,\_) & \triangleq & s[a] & [T].n(s,\_,a) & \triangleq & s[a]\ . \end{array}$$

Extrema *If $(w_i)_{i\in I}$ is an indexed family of interaction structures, then*

$$\begin{array}{lcll} (\bigcup_i w_i).A(s) & \triangleq & (\Sigma\, i \in I)\, w_i.A(s) & = s \mathrel{\epsilon} (\bigcup_i w_i).A \\ (\bigcup_i w_i).D\big(s,(i,a)\big) & \triangleq & w_i.D(s,a) & \\ (\bigcup_i w_i).n\big(s,(i,a),d\big) & \triangleq & w_i.n(s,a,d) & \end{array}$$

and

$$\begin{array}{lcll} (\bigcap_i w_i).A(s) & \triangleq & (\Pi\, i \in I)\, w_i.A(s) & = s \mathrel{\epsilon} (\bigcap_i w_i.A) \\ (\bigcap_i w_i).D(s,f) & \triangleq & (\Sigma\, i \in I)\, w_i.D(s,a) & \\ (\bigcap_i w_i).n\big(s,f,(i,d)\big) & \triangleq & w_i.n\big(s,f(i),d\big)\ . & \end{array}$$

Composition *Suppose w_1 and w_2 are interaction structures $S_1 \rightarrow Fam^2(S_2)$ and $S_2 \rightarrow Fam^2(S_3)$; define a structure $w_1 \fatsemi w_2$, called the* sequential composition *of w_1 and w_2. having type $S_1 \rightarrow Fam^2(S_3)$ with components:*

$$\begin{array}{lcl}
A(s_1) & \triangleq & (\Sigma\, a_1 \in A_1(s_1)) \\
 & & (\Pi\, d_1 \in D_1(s_1, a_1))\, A_2(s_1[a_1/d_1]) \\
D\big(s_1, (a_1, f)\big) & \triangleq & (\Sigma\, d_1 \in D_1(s_1, a_1))\, D_2\big(s[a_1/d_1], f(d_1)\big) \\
n\big(s_1, (a_1, f), (d_1, d_2)\big) & \triangleq & s_1[a_1/d_1][f(d_1)/d_2] \ .
\end{array}$$

i.e. a command in $(w_1 \fatsemi w_2).A(s)$ is given by a command in $w_1.A(s)$, and a continuation f giving, for all responses d in $w_1.D(s,a)$ a command in $w_2.A(s[a/d])$. Note that $(w_1 \fatsemi w_2).A = w_1{}^\circ(w_2.A)$.

Unit

$$\begin{array}{lcl}
\mathsf{skip}.A(s) & \triangleq & \{*\} \\
\mathsf{skip}.D(s, \_) & \triangleq & \{*\} \\
\mathsf{skip}.n(s, \_, \_) & \triangleq & s \ .
\end{array}$$

These operations satisfy the expected laws:

Proposition 3

$$\begin{array}{lcl}
\mathsf{skip}^\circ & = & \mathsf{skip}\ ; \\
\langle T \rangle^\circ & = & \langle T^\circ \rangle\ ; \\
[T]^\circ & = & [T^\circ]\ ; \\
(\bigcup_i w_i)^\circ & = & \bigcup_i (w_i{}^\circ)\ ; \\
(\bigcap_i w_i)^\circ & = & \bigcap_i (w_i{}^\circ)\ ; \\
(w_1 \fatsemi w_2)^\circ & = & w_1{}^\circ \fatsemi w_2{}^\circ\ ; \\
(w^\circ)^\perp & = & \mathsf{C} \cdot w^\circ \cdot \mathsf{C} \qquad \textbf{\textit{(only classically)}}\ .
\end{array}$$

Proof: Routine. Note that though to define the relation T° requires use of equality, one can define the predicate transformers $\langle T^\circ \rangle$ and $[T^\circ]$ without it. For the last point, we have constructively that

$$s \in (w^\circ)^\perp(V) \quad \text{iff} \quad (\forall U \subseteq S)\ s \in w^\circ(U) \Rightarrow U \between V$$

which can be taken as the definition of the dual for an arbitrary monotonic predicate transformer. This variant is better behaved in a constructive setting, and classically equivalent to the $\mathsf{C} \cdot \_ \cdot \mathsf{C}$ definition.

□

In view of this proposition, we may regard interaction structures as concrete representations of monotone predicate transformers that support many useful operators of the refinement calculus. (Iteration will be dealt with in subsection 4.4 on the following page.) As a result, we allow ourselves to overload the name w of an interaction structure to mean also w°.

4.3.2 Factorization of interaction structures

It is worth observing that any interaction structure $w : S \rightarrow Fam^2(S'')$ is equal to the composition $\langle T_a \rangle \fatsemi [T_d]$ where $T_a : S \rightarrow Fam(S')$, $T_d : S' \rightarrow Fam(S'')$

and $S' = (\Sigma\, s \in S)\, w.A(s)$. The transition structures T_a (which "issues the command") and T_d (which "performs the command") are defined as follows:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
T_a.A & \triangleq & w.A & T_d.A\big((s,a)\big) & \triangleq & w.D(s,a) \\
T_a.n(s,a) & \triangleq & (s,a) & T_d.n\big((s,a),d\big) & \triangleq & w.n(s,a,d)
\end{array}
$$

This factorization should be compared with the normal form theorem for predicate transformers mentioned on page 15. Just as $\langle R_a \rangle \fatsemi [R_d]$ is a normal form for monotone predicate transformers, so (with transition structures replacing relations) it is a normal form for interaction structures.

In this connection, one can define a symmetric variant of the notion of interaction structure, consisting of two arbitrary sets S and S' with either (i) a pair of relations between them, or (ii) a pair of transition structures in opposite directions. We have used the name "Janus structure" for type-(ii) structures (based on transition structures in different directions). Markus Michelbrink has used the name "interactive game" for type-(i) structures. Michelbrink's work shows that these to be highly interesting structures. The relation they bear to monotone predicate transformers seems not unlike that the natural numbers bear to the (signed) integers.

4.4 Iteration

We now define the iterative constructs $\_^*$ and $\_^\infty$ on interaction structures.

4.4.1 Angelic iteration

Definition 9 *Let $w : S \to Fam^2(S)$; define*

$$
\begin{array}{lcl}
w^*.A & \triangleq & (\mu\, X : S \to Set) \quad (\lambda\, s \in S) \\
 & & \textbf{data}\ \ \textsc{exit} \\
 & & \qquad\quad \textsc{call}(a,f)\ \textit{where}\ \ a \in S(s) \\
 & & \qquad\qquad\qquad\qquad\qquad f \in (\Pi\, d \in D(s,a))\, X(s[a/d])
\end{array}
$$

$$
\begin{array}{lcl}
w^*.D(s, \textsc{exit}) & \triangleq & \textbf{data}\ \ \textsc{nil} \\
w^*.D\big(s, \textsc{call}(a,f)\big) & \triangleq & \textbf{data}\ \ \textsc{cons}(d,d') \quad \textit{where}\ \ d \in D(s,a) \\
 & & \qquad\qquad\qquad\qquad\quad d' \in D^*\big(s[a/d], f(d)\big)
\end{array}
$$

$$
\begin{array}{lcl}
w^*.n(s, \textsc{exit}, \textsc{nil}) & \triangleq & s \\
w^*.n\big(s, \textsc{call}(a,f), \textsc{cons}(d_0,d')\big) & \triangleq & w^*.n\big(s[a/d_0], f(d_0), d'\big)
\end{array}
$$

An element of $A^*(s)$ is a data-structure that can be interpreted as a program or strategy for the Angel, to issue commands and react to the Demon's responses to commands. Because the definition uses a least fixpoint, this program is well founded in the sense that the Angel eventually reaches an EXIT command.

Associated with each such program $p \in A^*(s)$, the set $D^*(s,p)$ and the function $n^*(s,p,\_)$ give the family of states in which it can exit. Elements of the former can be seen as paths from s through p, while the latter maps a path to its final state. An element of $D^*(s,p)$ is sometimes called a (finite and complete) run, log, trace, or history. Note that a trace is intrinsically finite.

Proposition 4 *For any interaction structure w on S, we have $w^{*\circ} = w^{\circ *}$.*

Proof: Easy inductive proof. □

To make formulas easier to read, we adopt Sambin's "$\lhd$" notation:

Definition 10 *If $w : S \to Fam^2(S)$, $s \in S$, and $U, V \subseteq S$, put:*

$$\begin{array}{lcl} s \lhd_w U & \triangleq & s \,\epsilon\, w^{*\circ}(U) \ ; \\ V \lhd_w U & \triangleq & V \subseteq w^{*\circ}(U) \ . \end{array}$$

This higher-order relation satisfies:

Lemma 4.2

1. *monotonicity: $s \lhd_w U$ and $U \subseteq V \Rightarrow s \lhd_w V$;*
2. *reflexivity: $s \,\epsilon\, U \Rightarrow s \lhd_w U$;*
3. *transitivity: $s \lhd_w U$ and $U \lhd_w V \Rightarrow s \lhd_w V$.*

Proof: This is a just a rewriting of the definition of a closure operator using the "$\lhd$" notation. ($w^{*\circ}$ is a closure operator by proposition 2, since $w^{*\circ} = w^{\circ *}$.) Note that this proof (that $w^{*\circ}$ is a closure operator) is entirely predicative.

□

4.4.2 Demonic iteration

We first recall the rules used in [19] to generate "state dependent" greatest fixpoints. Translated to our setting, if (A, D, n) is an interaction structure on S, we are allowed to form the family A^∞ of sets indexed by $s \in S$ using the following rules:

- formation rule: $$\frac{s \in S}{A^\infty(s) : Set} \ ;$$
- introduction rule: (setting up a coalgebra) $$\frac{X : S \to Set \qquad F : X \subseteq w^\circ(X) \qquad s \in S \qquad x \in X(s)}{\mathsf{Coiter}(X, F, s, x) \in A^\infty(s)} \ ;$$ *(recall that $F : X \subseteq w^\circ(X)$ means $F : (\Pi\, s)\, X(s) \to (\Sigma\, a)\, (\Pi\, d)\, X(s[a/d])$)*
- elimination rule: $$\frac{s \in S \qquad K \in A^\infty(s)}{\mathsf{Elim}(s, K) \in w^\circ(A^\infty)(s)} \ ;$$
- reduction rule: $$\begin{array}{rl} \mathsf{Elim}\big(s, \mathsf{Coiter}(X, F, s, x)\big) = & \big(a, (\lambda\, d)\, \mathsf{Coiter}\big(X, F, s[a/d], g(d)\big)\big) \\ & \text{where} \quad (a, g) = F(s, x) \quad . \end{array}$$

  (Here "$(a, k) = \ldots$" is how we indicate an implicit pattern matching.)

It should be noted ([19, p. 11]) that those rules (which require that a weakly final coalgebra for w°) are dual to the rules for inductive types. Roughly speaking, they are the coinductive analogue of Petersson and Synek's inductively defined treeset constructions, expressed with a specific destructor Elim.[7]

[7] Implicit in these rules is a certain "weak" impredicative existential quantifier, that permits the formation of the higher product $(\Sigma\, X : Set)\, A : Set$, but without the strong projections of the usual Sigma type. Instead, one has an elimination rule closer to that in traditional natural deduction. Such a "weak" quantifier is sometimes invoked in the analysis of abstract data types ([31]).

Definition 11 *Let $w : S \to \mathit{Fam}^2(S)$; define*

$$
\begin{array}{lcl}
w^\infty.A & \triangleq & (\nu\, X : S \to \mathit{Set}) \quad (\lambda\, s \in S) \\
 & & (\Sigma\, a \in w.A(s))\,(\Pi\, d \in w.D(s,a))\; X(s[a/d]) \\
 & \triangleq & A^\infty \\[1ex]
w^\infty.D & \triangleq & (\mu\, X : (\Pi\, s \in S)\, A^\infty(s) \to \mathit{Set}) \quad (\lambda\, s \in S, p \in A^\infty(s)) \\
 & & \textbf{data} \;\; \textsc{nil} \\
 & & \qquad\;\; \textsc{cons}(d, d') \quad \textit{where} \;\; (a,k) = \mathsf{Elim}(p) \\
 & & \qquad\qquad\qquad\qquad\qquad\;\; d \in D(s,a) \\
 & & \qquad\qquad\qquad\qquad\qquad\;\; d' \in X(s[a/d], k(d))
\end{array}
$$

$$
\begin{array}{lcl}
w^\infty.n(s, p, \textsc{nil}) & \triangleq & s \\
w^\infty.n\big(s, p, \textsc{cons}(d, d')\big) & \triangleq & w^\infty.n\big(s[a/d], k(d), d'\big) \\
 & & \quad \textit{where} \;\; (a,k) = \mathsf{Elim}(p)
\end{array}
$$

An element of $A^\infty(s)$ can be interpreted as a command-response program starting in state s and continuing for as many cycles as desired, perhaps forever. One can picture such a program as an infinite tree, in which control flows along a branch in the tree. An element of $D^\infty(s,p)$ is a finite sequence of responses that may be returned to the agent running the program p; and $n^\infty(s,p,t)$ is the state obtained after the finite response sequence t has been processed.

Proposition 5 *For any interaction structure w on S, we have $w^{\infty\circ} = w^{\circ\infty}$.*

Proof: Let $U \subseteq S$:

- $w^{\infty\circ}(U) \subseteq w^{\circ\infty}(U)$: since $w^{\circ\infty}(U)$ is the greatest fixpoint of $U \cap w^\circ(\_)$, it suffices to show that $w^{\infty\circ}(U)$ is a post-fixpoint for the same operator, *i.e.* that $w^{\infty\circ}(U) \subseteq U \cap w^\circ\big(w^{\infty\circ}(U)\big)$.
  Let $s \mathrel{\epsilon} w^{\infty\circ}(U)$; this implies that there is some $p \in A^\infty(s)$ s.t.

  $$\big(\forall t \in D^\infty(s,p)\big)\; s[p/t] \mathrel{\epsilon} U \ .$$

  In particular, for $t = \textsc{nil}$, we have $s[p/\textsc{nil}] = s \mathrel{\epsilon} U$.

  We now show that $s \mathrel{\epsilon} w^\circ\big(w^{\infty\circ}(U)\big)$. Let $\mathsf{Elim}(p)$ be of the form (a_0, k). We claim that $\big(\forall d \in D(s,a_0)\big)\; s[a_0/d] \mathrel{\epsilon} w^{\infty\circ}(U)$: if $d \in D(s,a_0)$, we have $k(d) \in A^\infty(s[a_0/d])$ and $\textsc{cons}(d,d') \in D^\infty\big(s,(a_0,k)\big)$ for any d' in $D^\infty\big(s[a_0/d],k(d)\big)$. This implies (because $s \mathrel{\epsilon} w^{\infty\circ}(U)$) that

  $$s[a_0/d][k(d)/d'] \quad = \quad s[(a_0,k)/\textsc{cons}(d,d')] \quad \epsilon \quad U$$

  which completes the proof.

- $w^{\circ\infty}(U) \subseteq w^{\infty\circ}(U)$: let $s \mathrel{\epsilon} w^{\circ\infty}(U)$;
  we need to find a $p \in A^\infty(s)$ s.t. $\big(\forall t \in D^\infty(s,p)\big)\; s[p/t] \mathrel{\epsilon} U$. By the introduction rule for A^∞, it suffices to find a coalgebra $(X : S \to \mathit{Set}\,,\, F)$ with $F \in X \subseteq w^\circ X$.
  $X \triangleq w^{\circ\infty}(U)$ together with the function F coming from the coinductive rule $w^{\circ\infty} \subseteq \mathsf{skip} \cap (w^\circ \fatsemi w^{\circ\infty}) \subseteq w^\circ \fatsemi w^{\circ\infty}$ is such a coalgebra.
  This provides us with an element $\mathsf{Coiter}\big(X,F,s,x\big) \in A^\infty(s)$ where x is the proof that $s \mathrel{\epsilon} w^{\circ\infty}(U)$.

We will show the following: "for all states s, for all programs p generated by this coalgebra, for all responses t to p, we have $s[p/t] \mathrel{\epsilon} U$". More precisely, we will prove:

$$\Big(\forall s\,,\ \forall x \in X(s)\,,\ \forall t \in D^\infty\big(s, p(s,x)\big)\Big) \quad s[p(s,x)/t] \mathrel{\epsilon} U$$

where $p(s,x) = \mathsf{Coiter}(X, F, s, x)$.
We work by induction on the structure of t.

base case: if $t = \textsc{nil}$, then $s[p(s,x)/\textsc{nil}] = s$, and we have $s \mathrel{\epsilon} U$ since $s \mathrel{\epsilon} X = w^{\circ\infty}(U) \subseteq U$.

induction case: if $t = (d_0, t')$, then $s[p(s,x)/(d_0,t')] = s[a_0/d_0][k(d_0)/t']$ where $\mathsf{Elim}\big(p(s,x)\big) = (a_0, k)$. By the reduction rule for coinduction, we have; if x is of the form (a_0, f):

$$\mathsf{Elim}\,\mathsf{Coiter}(X,F,s,x) = \big(a_0, (\lambda\, d_0)\,\mathsf{Coiter}\big(X, F, s[a_0/d_0], f(d_0)\big)\big)$$

Therefore, $k(d_0) = p\big(s[a_0/d_0], f(d_0)\big)$, and we obtain the result by applying the induction hypothesis for $s[a_0/d_0]$, $f(d_0) \in X(s)$ and $t' \in D^\infty\big(s, p(s[a_0/d_0], f(d_0))\big)$.

□

Corollary 2 *For any interaction structure w, we have $w^{\perp^{\infty\circ}} = w^{\bullet\infty}$.*

Proof: Direct from lemma 4.1 and proposition 5. □

Just as for w^* and $\lhd$, we introduce the following notation:

Definition 12 *If w is an interaction structure on S, put:*

$$\begin{array}{rcl} s \ltimes_w U & \triangleq & s \mathrel{\epsilon} w^{\perp^{\infty}}(U)\ ; \\ V \ltimes_w U & \triangleq & V \between w^{\perp^{\infty}}(U)\ . \end{array}$$

4.5 Clients, servers and their interaction

In the vast majority of cases, there are only two kinds of program one is called upon to write: in programming terminology, those are called client programs and server programs. For background, see [40]. Clients and servers are agents on opposite sides of a service interface, sometimes also called a resource interface. The service may be, for example, to store values in addressable memory cells, or disk sectors. The client obtains or uses the service, the server provides it. In general terms, the behavior of an agent following a client program is to issue commands across the interface, and then use the responses to steer control to the right continuation point in the program, iterating through some finite number of command-response cycles until eventually reaching an exit point in the program. On the other hand, the behavior of an agent following a server program is to wait passively for a command, perform it and respond appropriately, for as many command-response cycles as required by the client.

The programming terminology of "clients" and "servers" is connected with the angelic and demonic forms of iteration described above in section 4.4. The client issues requests or commands, and the server performs them and responds to the client with a sequence of results, one for each issued command. Each request, its performance, and the response to it constitutes a *command-response cycle.* From the client's perspective, we may think of the performance of the request as an atomic event that occurs sometime between issuing the request and receiving the response. The server changes state, as it were "in a trice".

A server may have many clients. As when someone is operating a till in a supermarket, we may arrange (or simulate in various ways) that a client has the exclusive attention of a server, cycling through the purchase of several items by a single client, until the trolley is empty, the customer pays, and an entire *transaction*, consisting of many cycles is complete. Then the next customer in the queue comes forward. The number of cycles is at the discretion of the client. In essence what is happening here is that the server performs an entire transaction program (whose execution consists of several cycles) which we can view as a single composite command. The response to this composite command is a record or trace of responses to the individual commands: as it were, the receipt handed to the supermarket customer when the transaction is complete. However, what is important is that the transactions *appear* to take place in a total order. Outside of supermarkets, there are ways of processing transactions such that several transactions can be in progress, and their commitment is scheduled to optimize either throughput or response time. Essentially, starting a transaction is not something visible, and one can always pretend that transactions are started the instant before they are committed.

To describe clients and servers only in such a mechanistic way is however to miss something important. A client or server program is written to accomplish some purpose, or to fulfill an intention. The purpose or intention is expressed by a specification, ideally a formal specification that can be handled by a machine and used in verification. The crucial question is: what are the logical forms of the specifications of client and server programs? The interest of dependent type theory as a framework for developing programs is that one may hope, by exploiting the expressive power of the type system, to express specifications formally and yet with full precision. One may then harness decidable type-checking to guide the development of programs to meet those specifications.

Let us attempt to answer this question. What follows is merely an attempt to summarize experience of reading and writing specifications for both client and server programs.

Suppose w describes an interface; a *client program* is specified by a pair:

$\mathsf{Init} \subseteq S$**:** a predicate describing initial states in which the program is required to work. (In other states the program need not even terminate.) The user of the program is obliged to ensure that the initial predicate holds before running the program.

$\mathsf{Next} \subseteq S \times S$**:** a relation defined between initial states and final states. The value of Next for states outside Init is irrelevant: the behavior of the program is unspecified. Very often (but not always) this relation has the simpler "rectangular" form $\mathsf{Init} \times \mathsf{Goal}$ for some $\mathsf{Goal} \subseteq S$; meaning that the goal predicate does not depend on the initial state.

A client program satisfying such a specification is in essence a constructive proof that $\mathsf{Init} \subseteq \{\, s \in S \mid s \lhd_w \mathsf{Next}(s) \,\}$. When the Next relation happens to be of the form $\mathsf{Init} \times \mathsf{Goal}$, this takes the simpler form

$$\mathsf{Init} \quad \lhd_w \quad \mathsf{Goal} \; .$$

If we have such a proof, and the interface is in a state s such that initial predicate Init holds, then we can use the proof as a guide or strategy to bring about a state in which the goal predicate $\mathsf{Next}(s)$ holds, if only we are provided with a server that responds to all our requests.

As for server programs, the situation is the following: again, let w describe the interface. A *server program* is usually described by a pair of predicates:

$\mathsf{Init} \subseteq S$**:** a non-empty set which describes the allowed initial states of the service.

$\mathsf{Inv} \subseteq S$**:** a predicate that holds initially and is maintained by the server.

> **Remark.** By symmetry with specifications $\mathsf{Init} \subseteq \{\, s \in S \mid s \lhd_w \mathsf{Next}(s) \,\}$, where the relation Next is not necessarily rectangular, one may also consider server specifications of the form $\mathsf{Init} \between \{\, s \in S \mid s \ltimes_w \mathsf{Next}(s) \,\}$. At first sight the general case seems to have no counterpart in practice. However, if Next is actually a simulation relation one can express a certain kind of recoverability with a specification of this more general form. (This is connected with the discussion of localization at on page 45.)

A program satisfying such a specification is in essence a constructive proof that Init overlaps with the weakest post-fixpoint (invariant) of $w^\perp$ included in Inv. That is to say, it yields a state together with a proof that the state belongs to both the initial predicate and that invariant. Recall lemma 4.1 that if w is given by an interaction structure

$$U \subseteq w^\perp(U) \quad \Leftrightarrow \quad (\forall s \in U)\big(\forall a \in A(s)\big)\ \big(\exists d \in D(s,a)\big)\ s[a/d] \in U \; .$$

In other words the Demon is never deadlocked, but can always respond to any legal command, and moreover in such a way that the invariant continues to hold in the new state.

Note that a direct consequence of lemma 3.1 is that any invariant can be written in the form $(w^\perp)^\infty(V)$. The predicate V need not itself be an invariant, but can be weaker than the actual invariant $(w^\perp)^\infty(V)$, and so *a fortiori* is maintained by the server program.

To summarise, a server specification takes the form $\mathsf{Init} \between {w^\perp}^\infty(\mathsf{Inv})$ where Inv is a predicate guaranteed to hold before and after every step. Using the $\ltimes$ notation, this gives:

$$\mathsf{Init} \ltimes_w \mathsf{Inv} \; .$$

Interaction between client and server programs. What happens when we put a client and a server program together, and run the former "on" the latter? The answer is connected with the compatibility rule in Sambin's formalization of basic topology.

Suppose that in some state s of a common interface w, we have a client program P that can be run to bring about a goal predicate U (*i.e.* $s \lhd U$), and a server program K that maintains a predicate V (*i.e.* $s \ltimes V$). When all internal calculation has been carried out, the client program P will have been brought into one of two forms: either $(\textsc{call}(a,f),g)$ where

$$\begin{array}{l} a \in A(s) \\ f \in (\Pi\, d \in D(s,a))\, A^*(s[a/d]) \\ g \in \big(\Pi\, (d_0, d') \in D^*\big(s, \textsc{call}(a,f)\big)\big)\;\; s[a/d][f(d)/d'] \mathrel{\epsilon} U \;\; , \end{array}$$

or $(\textsc{exit}, h)$ where $h(\textsc{exit})$ is a proof that $s \mathrel{\epsilon} U$. On the other hand, if (K, l) is the *server* program, then $\mathsf{Elim}(s, K)$ has the form (r, k) where

$$\begin{array}{l} r \in (\Pi\, a \in A(s))\, D(s,a) \\ k \in (\Pi\, a \in A(s))\, (A^\perp)^\infty(s[a/r(a)]) \\ l \in \big(\Pi\, t \in (D^\perp)^\infty(s,K)\big)\, s[K/t] \mathrel{\epsilon} V \;\; . \end{array}$$

For any $U, V \subseteq S$, we define an *execution* function with the type

$$\mathtt{exec}_{U,V}\big((s,P,K) \mathrel{\epsilon} w^*(U) \between w^{\perp^\infty}(V)\big) \quad \in \quad U \between w^{\perp^\infty}(V)$$

by means of the following clauses:

$$\begin{array}{lcl} \mathtt{exec}_{U,V}\big(s,\, (\textsc{exit}, h),\, (K,l)\big) & \triangleq & (s, h(\textsc{exit}), K) \\ \mathtt{exec}_{U,V}\big(s,\, (\textsc{call}(a,f), g),\, (K,l)\big) & \triangleq & \end{array}$$

$$\begin{array}{lllcl} \mathbf{let} & (r,k) & = & \mathsf{Elim}(s,K) \\ & d & \triangleq & r(a) \\ & P' & \triangleq & f(d) \\ & g' & \triangleq & (\lambda\, d')\, g((d,d')) \\ & K' & \triangleq & k(a) \\ & l' & \triangleq & (\lambda\, t)\, l((a,t)) \\ \mathbf{in} & \multicolumn{4}{l}{\mathtt{exec}_{U,V}\big(s[a/d],\, (P', g'),\, (K', l')\big)} \end{array}$$

If we strip away the parameters and programs from this rule, we obtain

$$\frac{w^*(U) \between w^{\perp^\infty}(V)}{U \between w^{\perp^\infty}(V)}$$

that can immediately be recognized as Sambin's compatibility rule ([36]). In some sense this rule expresses the mechanics of interaction between client and server programs.

How does this rule apply to the formulas given above for the general form of client and server specifications? Suppose we have a client program satisfying the specification "$\mathsf{Init} \lhd_w \mathsf{Goal}$", and a server program satisfying the specification "$\mathsf{Init} \ltimes_w \mathsf{Inv}$". Then we can apply the execution function to get:

$$\frac{\mathsf{Init} \lhd \mathsf{Goal} \qquad \mathsf{Init} \ltimes \mathsf{Inv}}{\mathsf{Goal} \ltimes \mathsf{Inv}} \quad .$$

The real use of a client program is turn servers in a state that satisfies the precondition into servers in a state that satisfies the goal predicate.

Safety and Liveness. The concepts of partial and total correctness emerged from the investigations of Floyd, Dijkstra and Hoare into the foundations of specification and verification for sequential programming. A program is partially correct if it terminates only when it has attained the goal that it should, while it is totally correct if in addition it terminates whenever it should. In the late 70's, Lamport in [26] introduced the terms *safety* and *liveness* as the appropriate generalizations of these concepts to the field of concurrent programming. In concurrent programming a program interacts with its environment while it is running, rather than only when initialized or terminated. Informally, a safety property requires that "nothing bad" should occur during execution of a concurrent program. (A time can be associated with the violation of a safety property). On the other hand a liveness property requires that "something good" should occur (so that it is violated only at the end of time, as it were). These properties soon received formal definitions, in the case of safety by Lamport [27], and in the case of liveness by Alpern and Schneider [3].

These properties were defined in topological terms, with respect to the "Baire" space of infinite sequences of states. (The set of sequences sharing a common finite prefix is a basic neighborhood in this space). Briefly, a safety property was analyzed as a closed set of sequences, and a liveness property as a dense set (*i.e.* one intersecting with every non-empty open set). The properties were also expressed in terms of linear-time temporal logic, the idea being that a safety property asserts that something is (now and) *forever* the case, whereas a liveness property requires that something (now or) *eventually* takes place. For various reasons liveness is usually restricted to fairness properties in which the temporal modalities are nested at most twice. An example of a fairness requirement is so-called "strong" fairness, which requires that an event (state-change) of a certain kind occurs infinitely often providing that it is enabled infinitely often. A readable account of the rôle these concepts play in practical specification can be found in Lamport's book [28].

What can we say about these notions from the perspective of interaction structures? One thing that can be said with some confidence is that a safety property is an invariant. In basic topology, invariants represent closed sets. So this agrees with Lamport's topological analysis.

A liveness property on the other hand is merely a set of points which overlaps with every non-empty open set. It seems difficult to say anything interesting about liveness properties in general; but it may be easier when the properties are simple combinations of particular modalities such as "infinitely often" and "eventually always".

4.6 Product operations

We describe below two product operations on interaction structures. The first corresponds to an operation treated in the refinement calculus ([5]), while the second does not.

Synchronous tensor. Suppose w_1 and w_2 are two interaction structures on S_1 and S_2. We define $w_1 \otimes w_2$ on $S_1 \times S_2$:

$$\begin{array}{lcl}
(w_1 \otimes w_2).A((s_1, s_2)) & \triangleq & w_1.A(s_1) \times w_2.A(s_2) \\
(w_1 \otimes w_2).D((s_1, s_2), (a_1, a_2)) & \triangleq & w_1.D(s_1, a_1) \times w_2.D(s_2, a_2) \\
(w_1 \otimes w_2).n((s_1, s_2), (a_1, a_2), (d_1, d_2)) & \triangleq & (s_1[a_1/d_1], s_2[a_2/d_2])
\end{array}$$

The computational meaning of this operation is clear: one issues commands in each of a pair of interfaces, receives responses from them both, and they each move to their new state, simultaneously and atomically. Sometimes this kind of arrangement is called "ganging", or "lock-step synchronization".

The synchronous tensor corresponds to the following operation on predicate transformers (addition to propositions 3, 4 and 5):

$$(F_1 \otimes F_2)(R) = \bigcup_{U \times V \subseteq R} F_1(U) \times F_2(V)$$

which was used in [5] to model parallel execution of program components.

In combination with duality (definition 6), the synchronous tensor enjoys strong algebraic properties (see [23]).

Angelic product. Similarly, suppose w_1 and w_2 are two interaction structures on S_1 and S_2. We define $w_1 \odot w_2$ on $S_1 \times S_2$:

$$\begin{array}{lcl}
(w_1 \odot w_2).A((s_1, s_2)) & \triangleq & w_1.A(s_1) + w_2.A(s_2) \\
(w_1 \odot w_2).D((s_1, s_2), \mathsf{in}_0(a_1)) & \triangleq & w_1.D(s_1, a_1) \\
(w_1 \odot w_2).D((s_1, s_2), \mathsf{in}_1(a_2)) & \triangleq & w_2.D(s_2, a_2) \\
(w_1 \odot w_2).n((s_1, s_2), \mathsf{in}_0(a_1), d_1) & \triangleq & (s_1[a_1/d_1], s_2) \\
(w_1 \odot w_2).n((s_1, s_2), \mathsf{in}_1(a_2), d_2) & \triangleq & (s_1, s_2[a_2/d_2]) \ .
\end{array}$$

The computational meaning is again quite clear: a pair of interfaces is available to the Angel, who choose the one to use. This kind of arrangement is frequently found at the low-level interface of a program component, where instances of various resources are exploited, one at a time, to implement a higher-level interface. We call this kind of combination the "angelic product".

In terms of predicate transformers, the angelic product corresponds to

$$(F_1 \odot F_2)(R) = \bigcup_{\{s_1\} \times V \subseteq R} \{s_1\} \times F_2(V) \quad \cup \quad \bigcup_{U \times \{s_2\} \subseteq R} F_1(U) \times \{s_2\} \quad .$$

5 Morphisms

5.1 Linear simulations

We now consider what to take for morphisms between predicate transformers or their representation by interaction structures. The definition we adopt coincides with what is known as a "forward" simulation in the refinement calculus. As we will see in section 6, it is also connected with the definition of continuous relation in formal topology.

Let us therefore consider the case of (homogeneous) interaction structures, subscripted with "h" and "l" to distinguish the high and low level interfaces.

$$\begin{array}{lcccl} w_h & : & S_h & \rightarrow & \mathit{Fam}^2(S_h) \\ \downarrow & & & & \\ w_l & : & S_l & \rightarrow & \mathit{Fam}^2(S_l) \end{array}$$

As explained earlier, we view w_h and w_l as command-response interfaces over the state spaces S_h and S_l, where the command and response "dialects" are given by (A_h, D_h) and (A_l, D_l) respectively. Our intuition here is to think of a morphism as a systematic translation between the dialect for w_h and the dialect for w_l, which enables us to use a device supporting the interface (S_l, w_l) as if it were a device supporting the interface (S_h, w_h). That is, we should be able to translate high level A_h-commands into low level A_l-commands, and responses to the latter (low level D_l responses) back into high level D_h responses in such a way that the simulation of (S_h, w_h) by (S_l, w_l) can be indefinitely sustained.

It is often the case that several different low-level states can represent the same high-level state, so that the link between high-level states and low-level states can be represented by a function from the latter to the former (sometimes called an abstraction function, or refinement mapping). It is also sometimes the case that several different high-level states can be represented by the same low-level state. For such reasons, many people take the link between high and low level states to be a general relation, rather than a map one one direction or the other.

The question then is: how can we make this intuition of translation precise? The answer we propose is the following.

Definition 13 *Let $w_h : S_h \rightarrow \mathit{Fam}^2(S_h)$, and $w_l : S_l \rightarrow \mathit{Fam}^2(S_l)$. A* linear simulation *of (S_h, w_h) by (S_l, w_l) is a relation $R \subseteq S_h \times S_l$ which satisfies the following "sustainability" condition:*

$$\begin{array}{ll} \textit{If } (s_h, s_l) \mathrel{\epsilon} R, \textit{ then} & \\ \quad \forall a_h \in A_h(s_h) & \textit{– for all high-level commands } a_h \ldots \\ \quad \exists a_l \in A_l(s_l) & \textit{– there is a low-level command } a_l \textit{ s.t.} \\ \quad \forall d_l \in D_l(s_l, a_l) & \textit{– for all responses } d_l \textit{ to the low-level command} \ldots \\ \quad \exists d_h \in D_h(s_h, a_h) & \textit{– there is a response } d_h \textit{ to the command } a_h \textit{ s.t.} \\ \quad\quad \big(s_h[a_h/d_h], s_l[a_l/d_l]\big) \mathrel{\epsilon} R & \textit{– the simulation can be sustained.} \end{array}$$

We write $R : w_h \multimap w_l$ to mean R is a linear simulation from w_h to w_l.

In explanation of the qualification "linear", we have required a one-for-one intertranslation between the high and low-level interfaces. (We shall shortly introduce a notion of general simulation, allowing zero or non-zero low-level interactions for each high-level interaction.)

The formula above with its four nested quantifiers is perhaps a little daunting at first sight. Let's re-express it in a more compact form.

Lemma 5.1 *$R \subseteq S_h \times S_l$ is a linear simulation of (S_h, w_h) by (S_l, w_l) iff for all $s_h \in S_h$, and $a_h \in A_h(s_h)$, we have $R(s_h) \subseteq w_l\big(\bigcup_{d_h \in D_h(s_h, a_h)} R(s_h[a_h/d_h])\big)$.*

Proof: Simple formal manipulation. □

Remark. A linear simulation from w_h to w_l is itself an invariant for a certain relation transformer "$w_h \multimap w_l$".

Definition 14 *If w_h and w_l are interaction structures on S_h and S_l, define a new interaction structure on $S_h \times S_l$ with:*

$$\begin{array}{lcl}
A((s_h, s_l)) & \triangleq & (\Sigma\, f \in A_h(s_h) \to A_l(s_l)) \\
 & & (\Pi\, a_h \in A_h(s_h))\ \ D_l(s_l, f(a_h)) \to D_h(s_h, a_h) \\
D((s_h, s_l), (f, g)) & \triangleq & (\Sigma\, a_h \in A_h(s_h))\ \ D_l(s_l, f(a_h)) \\
n((s_h, s_l), (f, g), (a_h, d_l)) & \triangleq & (s_h[a_h/g(a_h, d_l)], s_l[f(a_h)/d_l])\ .
\end{array}$$

This concrete representation is merely the result of applying the axiom of choice to pull the quantifier alternation $(\Pi\,\_)\,(\Sigma\,\_)\,(\Pi\,\_)\,(\Sigma\,\_)$ into $(\Sigma\,\_)\,(\Pi\,\_)$ form. Notice that everything has a computational meaning: the commands are intertranslation functions, the responses are data outside the control of the simulation, and data is communicated between the high and low poles of a state-pair.

Classically, this interaction structure is (isomorphic to) the representation of the linear-logic implication from [23]. The corresponding tensor is the synchronous tensor $\otimes$ defined on page 31. (One can check that "$\otimes$" is left-adjoint to "$\multimap$".) It is interesting to remark that neither composition nor iteration of predicate transformers/interaction structures are used in the models of linear logic from [23, 24].

The following proposition gives a characterization of linear simulations as a subcommutativity property (point *2*).

Proposition 6 *The following are equivalent:*

1. *R is a linear simulation of (S_h, w_h) by (S_l, w_l);*
2. *$\langle R^{\sim}\rangle \fatsemi w_h \subseteq w_l \fatsemi \langle R^{\sim}\rangle$;*
3. *for all $U \subseteq S_h$, $s_h \in S_h$ we have $s_h \lhd_{w_h} U \Rightarrow R(s_h) \lhd_{w_l} R(U)$.*

Proof: *(the implication 2⇒1 requires the use of equality)*

1⇒2: we have to show that $s_l \mathrel{\epsilon} (R \fatsemi w_h)(U)$ implies $s_l \mathrel{\epsilon} (w_l \fatsemi R)(U)$.

$$\begin{array}{ll}
 & s_l \mathrel{\epsilon} R \fatsemi w_h(U) \\
\Leftrightarrow & \big\{ \text{ definition of } \fatsemi \big\} \\
 & (\exists s_h \in S_h)\ (s_h, s_l) \mathrel{\epsilon} R \text{ and } s_h \mathrel{\epsilon} w_h(U) \\
\Rightarrow & \big\{ \text{ definition of the predicate transformer } w_h \big\} \\
 & (\exists s_h)\quad (s_h, s_l) \mathrel{\epsilon} R \text{ and} \\
 & \qquad\quad \big(\exists a_h \in A_h(s_h)\big)\,\big(\forall d_h \in D_h(s_h, a_h)\big)\ s_h[a_h/d_h] \mathrel{\epsilon} U \\
\Rightarrow & \big\{ \text{ by lemma 5.1 } \big\} \\
 & (\exists s_h)\quad (s_h, s_l) \mathrel{\epsilon} R \text{ and} \\
 & \qquad\quad s_l \mathrel{\epsilon} w_l\big(\bigcup_{d_h} R(s_h[a_h/d_h])\big) \text{ and } \bigcup_{d_h} s_h[a_h/d_h] \subseteq U \\
\Rightarrow & \big\{ \ R = \langle R^{\sim}\rangle \text{ commutes with unions } \big\} \\
 & (\exists s_h)\quad (s_h, s_l) \mathrel{\epsilon} R \text{ and} \\
 & \qquad\quad s_l \mathrel{\epsilon} w_l R\big(\bigcup_{d_h} s_h[a_h/d_h]\big) \text{ and } \bigcup_{d_h} s_h[a_h/d_h] \subseteq U \\
\Rightarrow & \big\{ \text{ by monotonicity } \big\} \\
 & s_l \mathrel{\epsilon} w_l \fatsemi R(U)
\end{array}$$

2⇒1: suppose that $R \fatsemi w_h \subseteq w_l \fatsemi R$, and let $s_h \in S_h$ and $a_h \in A_h(s_h)$; we will show that $R(s_h) \subseteq w_l\big(\bigcup_{d_h \in D_h(s_h, a_h)} R(s_h[a_h/d_h])\big)$ and conclude using lemma 5.1.

Define $U \triangleq \bigcup_{d_h \in D_h(s_h, a_h)} \{s_h[a_h/d_h]\}$. (This where equality is needed.)

We certainly have that $s_h \mathrel{\epsilon} w_h(U)$ so that $R(s_h) \subseteq R \fatsemi w_h(U)$. By hypothesis, this implies that $R(s_h) \subseteq w_l \fatsemi R(U)$ which we had set out to prove. (Since R commutes with unions.)

The proof that *2*⇔*3* is straightforward.

□

The following is easy:

Proposition 7 *If w_1 and w_2 are interaction structures, the linear simulations of w_1 by w_2 are closed under arbitrary unions (including the empty union, so that there is always an empty simulation).*

Finally, the following shows that we have a poset enriched category.

Proposition 8

1. *The relational composition $(R_1 \mathbin{;} R_2)$ of two linear simulations is a linear simulation.*
2. *If w is an interaction structure on S, then $\mathsf{eq}_S : w \multimap w$.*
3. *Composition of linear simulations is monotone in both its arguments.*

We call this category **LinSim**.

Proof: Straightforward. □

The same proposition holds if we replace interaction structures with predicate transformers, and use point *2* from proposition 6 as the definition of simulation. We call this category **PT**.

> **Remark.** Of course to define a category, we need equality relations for the identity morphisms of this category. Without equality, we have a weaker structure, having merely an associative and monotone composition of morphisms.

A morphism is supposed to "preserve structure". What is the structure preserved by a simulation? The following observation suggests one answer.

Lemma 5.2 *If R is a simulation as above, the image of an invariant for w_h is an invariant for w_l, i.e. the image of a high-level invariant is a low-level invariant.*

Proof: simple application of proposition 6. □

> **Remark.** The notion of a linear simulation is already well-known in the literature of the refinement calculus (see for example [6]). There it is known as forward (or "downward") data refinement. In fact, in that setting one considers a more general notion, in which the relation (which may be identified with a disjunctive predicate transformer) is generalized to a "right-moving" predicate transformer:
>
> **Definition 15** *If F_h and F_l are transformers, and if $P : Pow(S_l) \to Pow(S_h)$, then F_h is said to be data-refined through P by F_l if*
>
> $$P \mathbin{;} F_h \subseteq F_l \mathbin{;} P \ .$$
>
> *If P commutes with arbitrary unions, then the refinement is said to be "forward", whereas if P commutes with arbitrary intersections, the refinement is said to be "backward".*
>
> In the setting of impredicative higher-order logic one can prove that the predicate transformers that commute with arbitrary unions are precisely those of the form $\langle Q \rangle$ for some relation Q, and those that commute with arbitrary intersections are precisely those of the form $[Q]$. It follows that a linear simulation is a forward data-refinement. It is natural to wonder whether one can give a predicative analysis of backward data refinement, akin to that we have given of forward data refinement.

5.2 Monads and general simulations

When a high-level interface is implemented on top of a low-level, less abstract interface, it is rare that a single high-level command (for example: record this data as a file in such and such a directory) can be translated to a *single* low-level command. Instead, several interactions across the low-level interface (reading and writing disk sectors) are usually required before the high-level operation can be completed. In essence, what we are going to do is make the notion of simulation more flexible and applicable by moving to the Kleisli category for a certain monad.

There are at least three monads of interest: the reflexive closure, the transitive closure and the reflexive/transitive closure.

RC The functor $RC(F) = \mathsf{skip} \cup F$ is monadic. A morphism in the Kleisli category from (S_1, F_1) to (S_2, F_2) is a linear simulation of (S_1, F_1) by $\big(S_2, RC(F_2)\big)$, which we call an *affine simulation* of (S_1, F_1) by (S_2, F_2). A step in (S_1, F_1) need not make use of a step in (S_2, F_2).

RTC $\_^*$ is monadic. A morphism in the Kleisli category from (S_h, F_h) to (S_l, F_l) is a linear simulation of (S_h, F_h) by (S_l, F_l^*), which we call a *general simulation* of (S_h, F_h) by (S_l, F_l). A step in (S_h, F_h) may make use of any number of steps in (S_l, F_l).

TC The functor $F^+ = F \fatsemi F^*$ is monadic. A morphism in its Kleisli category is a linear simulation of (S_h, F_h) by $\big(S_l, {F_l}^+\big)$. It translates high-level commands to low level programs that run for at least one step.

Proposition 9 *$RC(\_)$, $\_^+$ and $\_^*$ are monads in* **LinSim** *and* **PT**. *We call the Kleisli category of $\_^*$ the category of general simulations and interaction structures:* **GenSim**. *We write $R : w_h \to w_l$ for morphisms in this category (i.e. $w_h \to w_l$ is a synonym for $w_h \multimap w_l^*$).*

Proof: We will work with interaction structures; the case of predicate transformer is very similar. Moreover, we only treat the case of the $\_^*$ functor; the other cases being similar.

Recall that an endofunctor M on a category $\mathcal{C}$ is a monad (in triple form) if we have the following:

- an operation $\_^\sharp$ taking any $f : \mathcal{C}\big[A, M(B)\big]$ to an $f^\sharp : \mathcal{C}\big[M(A), M(B)\big]$;
- for any object A of $\mathcal{C}$, a morphism $\eta_A : \mathcal{C}\big[A, M(A)\big]$

such that:

1. $\eta \fatsemi f^\sharp = f$;
2. ${\eta_A}^\sharp = \mathsf{id}_{M(A)}$;
3. $(f \fatsemi g^\sharp)^\sharp = f^\sharp \fatsemi g^\sharp$.

It is trivial to check that $\mathsf{eq} : w \multimap w^*$; and the next proposition will show that if R is a linear simulation of w_h by w_l^* then R is a linear simulation of w_h^* by w_l^*. Thus we can put: $R^\sharp \triangleq R$ and $\eta_{(S,w)} \triangleq \mathsf{eq}_S$.

□

Lemma 5.3 *Let w_h and w_l be two interaction structures on the sets S_h and S_l; let R be a relation on $S_h \times S_l$. The following are equivalent:*

1. *R is a linear simulation $w_h \multimap w_l^*$;*
2. *for any $s_h \in S_h$ and $a_h \in A_h(s_h)$:*

$$R(s_h) \lhd_{w_l} \bigcup_{d_h \in D_h(s_h,a_h)} R\big(s_h[a_h/d_h]\big) \ ;$$

3. *for any $s_h \in S_h$ and $a'_h \in A_h^*(s_h)$:*

$$R(s_h) \lhd_{w_l} \bigcup_{d'_h \in D_h^*(s_h,a'_h)} R\big(s_h[a'_h/d'_h]\big) \ .$$

Proof: In turn:

1⇔*2*: simple consequence of proposition 6.

3⇒*2*: follows from the observation that eq_{S_h} is a linear simulation $w_h \multimap w_h^*$.

2⇒*3*: let $s_h \in S_h$ and $a'_h \in A_h^*(s_h)$; we do the proof by induction on a'_h:

base case if $a'_h = \textsc{exit}$, then we only need to show that $R(s_h) \lhd_{w_l} R(s_h)$, which is trivially true since $\mathsf{skip} \subseteq w_l^*$.

induction case if a'_h is of the form $\textsc{call}(a_h, f_h)$, then we have:

- $R(s_h) \lhd \bigcup_{d_h} R\big(s_h[a_h/d_h]\big)$; *(by point 1⇔2 of this lemma)*
- for any $d_h \in D_h(s_h, a_h)$, by induction hypothesis, we have:

$$R\big(s_h[a_h/d_h]\big) \quad \lhd_{w_l} \quad \bigcup_{d'_h} R\big(s_h[a_h/d_h][f_h(d_h)/d'_h]\big)$$

  where $d'_h \in D_h^*(s_h[a_h/d_h], f_h(d_h))$

- since the RHS is a subset of $\bigcup_{d_h,d'_h} R\big(s_h[\textsc{call}(a_h,f_h)/(d_h,d'_h)]\big)$ we can conclude (by monotonicity) that

$$R\big(s_h[a_h/d_h]\big) \quad \lhd_{w_l} \quad \bigcup_{d'_h} R\big(s_h[a'_h/d'_h]\big)$$

  which, by transitivity, implies

$$R(s_h) \quad \lhd_{w_l} \quad \bigcup_{d'_h} R\big(s_h[a'_h/d'_h]\big)$$

□

Corollary 3 *We have: R is a linear simulation $w_h \multimap w_l^*$ iff R is a linear simulation $w_h^* \multimap w_l^*$.*

5.3 Saturation, equality of morphisms

We have argued that the category **GenSim** serves as a model for component based programming. However, the notion of equality on morphisms is still too strong. It may be that two general simulations differ extensionally though they still have the same potential, or "simulative power".

Definition 16 *If R is a general simulation from w_h to w_l, we define the following relation $\overline{R}$ (the saturation of R) on $S_h \times S_l$:*

$$(s_h, s_l) \mathrel{\epsilon} \overline{R} \quad \triangleq \quad s_l \mathrel{\epsilon} w_l^* \cdot R(s_h) \ .$$

This amounts to considering instead of functions $R : S_h \rightarrow Pow(S_l)$, functions $R : S_h \rightarrow Sat(w_l)$, where $Sat(w_l)$ is the collection of w_l-saturated predicates. (See lemma 3.1.)

The intuition behind saturation is the following. Suppose R is a relation between low level states S_l and high level states S_h. The saturation of R is a relation which allows "internal" or "hidden" low level interaction. To simulate a high level state s_h by a low level state s_l, it is permissible that the Angel has a program that constrains interactions starting in s_l to terminate in states that simulate s_h.

We have:

Proposition 10 *Let R be a general simulation of w_h by w_l, then $\overline{R}$ is also a general simulation of w_h by w_l.*

Proof: According to lemma 5.3, we need to show $\overline{R}(s_h) \lhd \bigcup_{d_h} \overline{R}\big(s_h[a_h/d_h]\big)$.

By lemma 5.3, we have $R(s_h) \lhd \bigcup_{d_h} R\big(s_h[a_h/d_h]\big)$ and since w_2^* is a closure operator, we have

$$w_2^*\big(R(s_h)\big) \equiv \overline{R}(s_h) \quad \lhd \quad \bigcup_{d_h} R\big(s_h[a_h/d_h]\big) \ .$$

For any d_h, $R\big(s_h[a_h/d_h]\big) \lhd R\big(s_h[a_h/d_h]\big)$ which implies (still because w_2^* is a closure operator)

$$R\big(s_h[a_h/d_h]\big) \quad \lhd \quad w_2^*\Big(R\big(s_h[a_h/d_h]\big)\Big) \equiv \overline{R}(s_h[a_h/d_h]) \ .$$

Since the above is true for any d_h, it implies that

$$\bigcup_{d_h} R\big(s_h[a_h/d_h]\big) \quad \lhd \quad \bigcup_{d_h} \overline{R}\big(s_h[a_h/d_h]\big) \ .$$

We get the result by transitivity.

□

Thus, saturation provides us with an appropriate "normalization" operation when comparing general simulations: to compare two simulations, compare their normal forms. So we put:

Definition 17 *Let R_1, R_2 be two general simulations of w_h by w_l; we say that*

- *R_2 is* stronger *than R_1 (written $R_1 \sqsubseteq R_2$) if $\overline{R_1} \subseteq \overline{R_2}$;*

- R_1 *is* equivalent *to* R_2 *(*$R_1 \approx R_2$*) if* $R_1 \sqsubseteq R_2$ *and* $R_2 \sqsubseteq R_1$*.*

The following is trivial: (point *3* follows from proposition 10).

Lemma 5.4 *We have:*

1. $\sqsubseteq$ *is a preorder on the collection of general simulations from* w_h *to* w_l*;*
2. $\approx$ *is an equivalence relation;*
3. $\overline{R}$ *is (extensionally) the largest relation in the equivalence class of* R*;*
4. *the operation* $R \mapsto \overline{R}$ *is a closure operation.*

We can now conclude this section:

Proposition 11 $(\mathbf{GenSim}, \sqsubseteq)$ *is a poset enriched category.*

Proof: The only thing we need to check is that composition is monotonic in both its arguments.[8]

Let R_1, R_2 be two simulations of w_h by w_m and Q_1, Q_2 two simulations of w_m by w_l such that $R_1 \sqsubseteq R_2$ and $Q_1 \sqsubseteq Q_2$. Suppose moreover that $s_h \in S_h$; we need to show that $\overline{R_1 \fatsemi Q_1}(s_h) \subseteq \overline{R_2 \fatsemi Q_2}(s_h)$:

- we have $\overline{R_1}(s_h) \subseteq \overline{R_2}(s_h)$ because $R_1 \sqsubseteq R_2$;
- we also have $\big(Q_1 \fatsemi \overline{R_1}\big)(s_h) \lhd_l \big(Q_2 \fatsemi \overline{R_2}\big)(s_h)$:

  let $s_l \epsilon \big(Q_1 \fatsemi \overline{R_1}\big)(s_h)$, *i.e.* $(s_m, s_l) \epsilon Q_1$ for some s_m s.t. $(s_h, s_m) \epsilon \overline{R_1}$. We will show that $s_l \lhd_l \big(Q_2 \fatsemi \overline{R_2}\big)(s_h)$:

  - $Q_2(s_m) \subseteq \big(Q_2 \fatsemi \overline{R_2}\big)(s_h)$ since $s_m \epsilon \overline{R_1}(s_h) \subseteq \overline{R_2}(s_h)$;
  - $s_l \epsilon \overline{Q_2}(s_m)$ because $Q_1 \sqsubseteq Q_2$ and $s_l \epsilon \overline{Q_1}(s_m)$; *(since $s_l \epsilon Q_1(s_m)$)*
  - so by monotonicity, $s_l \epsilon \big(w_l^* \fatsemi Q_2 \fatsemi \overline{R_2}\big)(s_h)$.
- From the last point, we get $\big(w_l^* \fatsemi Q_1 \fatsemi \overline{R_1}\big)(s_h) \subseteq \big(w_l^* \fatsemi Q_2 \fatsemi \overline{R_2}\big)(s_h)$;
- for any simulation $R : w \multimap w'^*$, we have $\big(w'^* \fatsemi R \fatsemi w^*\big)(U) = \big(w'^* \fatsemi R\big)(U)$:

  $\subseteq$: because $\big(R \fatsemi w^*\big)(U) \subseteq \big(w'^* \fatsemi R\big)(U)$ and w'^* is a closure operator;

  $\supseteq$: $\mathsf{skip} \subseteq w^* \Rightarrow w'^* \fatsemi R \subseteq w'^* \fatsemi R \fatsemi w^*$.
- So we can conclude:

$$\overline{R_1 \fatsemi Q_1}(s_h) \equiv \big(w_l^* \fatsemi Q_1 \fatsemi R_1\big)(s_h) \subseteq \big(w_l^* \fatsemi Q_2 \fatsemi R_2\big)(s_h) \equiv \overline{R_2 \fatsemi Q_2}(s_h)\ .$$

□

6 The link with formal topology

Our title mentions both programming and formal topology. We now (at last) turn to the topological meaning of our constructions. We start by recalling the most basic notions of formal topology.

[8] This result, together with all the required lemmas has been formally checked using the Agda system.

6.1 Formal and basic topology

The aim of formal topology was to develop *pointfree topology* in a fully constructive (*i.e.* predicative) setting. Motivations for pointfree topology can be found in [25]. Briefly, pointfree topology studies the properties of the lattice of open sets of a topology, without ever mentioning points (hence the name). Many traditional topological theorems are classically equivalent to a pointfree version that can be proved constructively without the axiom of choice. Example of such theorems include Hahn-Banach theorem, Heine-Borel theorem, or various representation theorems (such as Stone's). The idea is thus to factor out all non-constructive methods into the proof that the pointfree version is equivalent to the traditional theorem.

Basic topology amounts to removing the condition of distributivity of the lattice of open sets. The result is a very concise and elegant structure which, surprisingly enough, still contains the basic notions of topology (closed sets, open sets and continuity). It is the basis of a modular approach to formal topology in that one can add exactly what is needed in order to understand a particular property.

Introductions to the subject can be found in [25, 43, 34, 38, 37, 15].

6.1.1 Basic topologies

See [14] for details.

Definition 18 *A* basic topology *is a set S together with two predicate transformers $\mathcal{A}$ and $\mathcal{J}$ on S such that:*

- *$\mathcal{A}$ is a closure operator;*
- *$\mathcal{J}$ is an interior operator;*
- *$\mathcal{A}$ and $\mathcal{J}$ are* compatible*:* $\dfrac{\mathcal{A}(U) \between \mathcal{J}(V)}{U \between \mathcal{J}(V)}$ *for all $U, V \subseteq S$.*

The set S is intended to represent a base of the topology; and so, an element $s \in S$ will be called a *formal basic open.* A subset U of S is called *open* when $U = \mathcal{A}(U)$; and a subset V of S is called *closed* when $V = \mathcal{J}(V)$.[9]

A minimal requirement is that open sets [resp. closed sets] form a sup lattice [resp. inf lattice]. This is indeed the case:

Lemma 6.1 *If $(U_i)_{i \in I}$ is family of open sets, define $\bigvee_i U_i = \mathcal{A}\big(\bigcup_{i \in I} U_i\big)$; the type of open sets with $\bigvee$ and $\cap$ is a lattice with all set-indexed sups.*

If $(V_i)_{i \in I}$ is family of closed sets, define $\bigwedge_i V_i = \mathcal{J}\big(\bigcap_{i \in I} V_i\big)$; the type of closed sets with $\bigwedge$ and $\cup$ is a lattice with all set-indexed infs.

However, these lattices are generally speaking not distributive. (We will see a way to add distributivity in section 6.1.3.) As a consequence there is no notion of *point* in basic topology.[10]

[9] No mistakes: a formal open is *closed* in the sense of $\mathcal{A}$; and a formal closed is open in the sense of $\mathcal{J}$! See [35] for the justification.

[10] More precisely, without distributivity the notion of a point cannot be distinguished from that of a closed subset!

6.1.2 Formal continuity

See [15] for details.

Since a continuous function from $(S_1, \mathcal{A}_1, \mathcal{J}_1)$ to $(S_2, \mathcal{A}_2, \mathcal{J}_2)$ should map open sets in $(S_2, \mathcal{A}_2, \mathcal{J}_2)$ to open sets in $(S_1, \mathcal{A}_1, \mathcal{J}_1)$, it cannot be represented directly by a function from S_2 to S_1. A continuous function has to be represented by a relation between S_1 and S_2. If $f \subseteq S_1 \times S_2$ represents such a continuous function, the intuitive, concrete meaning of $(s_1, s_2) \in f$ is thus "$s_1 \subseteq f^{-1}(s_2)$", where s_1 and s_2 are basic opens.

Definition 19 *If $(S_1, \mathcal{A}_1, \mathcal{J}_1)$ and $(S_2, \mathcal{A}_2, \mathcal{J}_2)$ are basic topologies, and R a relation between S_1 and S_2; R is* continuous *if the two conditions hold:*

1. $R^{\sim}\big(\mathcal{A}_2(V)\big) \subseteq \mathcal{A}_1\big(R^{\sim}(V)\big)$;
2. $R\big(\mathcal{J}_1(U)\big) \subseteq \mathcal{J}_2\big(R(U)\big)$.

Equivalent characterizations are listed in [15]. It is worth noting that the two conditions are in general independent.

By definition, two continuous relations R and T from S_1 to S_2 are (topologically) equal if $\mathcal{A}(R^{\sim}s_2) = \mathcal{A}(T^{\sim}s_2)$ for all $s_2 \in S_2$. The main purpose of this definition is to remove dependency on the specific "base" of the topology considered.

Basic topologies and continuous relations with topological equality form a category which is called **BFTop**.

6.1.3 Convergent basic topologies

See [36] for details.

The above structure still lacks many properties found in "real" topologies; in particular, the binary infimum need not distribute over arbitrary suprema. One way to get distributivity is to add the following condition on the operator $\mathcal{A}$:

Definition 20 *Let $\mathcal{A}$ be a closure operator on a set S; write $U \downarrow V$ for the subset $\big\{s \mid (\exists s' \in U)\ s \in \mathcal{A}\{s'\}$ and $(\exists s'' \in V)\ s \in \mathcal{A}\{s''\}\big\}$. We say that $\mathcal{A}$ is* convergent *if the following holds:*

$$\frac{s \in \mathcal{A}(U) \qquad s \in \mathcal{A}(V)}{s \in \mathcal{A}(U \downarrow V)} \quad .$$

This condition is sometimes called *summability of approximations:* it gives a way to compute the intersection of two open sets from their representatives. If U and V represent the two open sets $\mathcal{A}(U)$ and $\mathcal{A}(V)$,[11] then $U \downarrow V$ represents the intersection $\mathcal{A}(U) \cap \mathcal{A}(V)$.

Lemma 6.2 *If $(S, \mathcal{A}, \mathcal{J})$ is a convergent basic topology, then its lattice of open sets is distributive.*

Proof: For any $U \subseteq S$, define $U^{\downarrow} = \big\{s \in S \mid (\exists s' \in U)\, s \in \mathcal{A}\{s'\}\big\}$. We have $U \downarrow V = U^{\downarrow} \cap V^{\downarrow}$.

[11] It is a trivial observation that a subset is open iff it is of the form $\mathcal{A}(U)$.

Let U be an open set and $(V_i)_{i\in I}$ a set-indexed family of open sets; *i.e.* we have $U = \mathcal{A}(U)$ and $V_i = \mathcal{A}(V_i)$ for all $i \in I$.

$$
\begin{array}{ll}
 & \bigvee_{i\in I} U \cap V_i \\
= & \big\{ \text{ U and the V_i's are open } \big\} \\
 & \bigvee_{i\in I} \mathcal{A}(U) \cap \mathcal{A}(V_i) \\
= & \big\{ \text{ convergence } \big\} \\
 & \bigvee_{i\in I} \mathcal{A}(U \downarrow V_i) \\
= & \big\{ \text{ definition of $\bigvee$ and easy lemma: $\mathcal{A} \bigcup \mathcal{A} = \mathcal{A} \bigcup$ } \big\} \\
 & \mathcal{A}\Big(\bigcup_{i\in I} U \downarrow V_i \Big) \\
= & \big\{ \text{ distributivity of $\cap$ and $\bigcup$ } \big\} \\
 & \mathcal{A}\Big(U^{\downarrow} \cap \bigcup_{i\in I} (V_i^{\downarrow}) \Big) \\
= & \\
 & \mathcal{A}\Big(U \downarrow (\bigcup_{i\in I} V_i) \Big) \\
= & \big\{ \text{ convergence } \big\} \\
 & \mathcal{A}(U) \cap \mathcal{A}\Big(\bigcup_{i\in I} V_i \Big) \\
= & \big\{ \text{ a union of open sets is an open set } \big\} \\
 & U \cap \bigvee_{i\in I} V_i
\end{array}
$$

which completes the proof that open sets do indeed form a frame.

□

Traditionally, formal topologies are also equipped with a *positivity predicate* called **Pos**. Its intuitive meaning is "$s \mathrel{\epsilon}$ **Pos** iff s is non-empty". This predicate was required to satisfy the *positivity axiom*: *(where $U^+ = U \cap \mathbf{Pos}$)*

$$\frac{s \mathrel{\epsilon} \mathcal{A}(U^+)}{s \mathrel{\epsilon} \mathcal{A}(U)}$$

which means that only positive opens really contribute to the topology.

The positivity predicate is now defined from $\mathcal{J}$: $\mathbf{Pos} \triangleq \mathcal{J}(S)$, and the positivity axiom is not required anymore. (Though it will hold in all examples with a real topological flavor.)

In a convergent basic topology, we can define the notion of point: (see [15])

Definition 21 *Let $(S, \mathcal{A}, \mathcal{J})$ be a convergent basic topology; a subset $\alpha \subseteq S$ is said to be a* point *if:*

1. *α is closed: $\alpha = \mathcal{J}(\alpha)$;*
2. *α is non-empty: $\alpha \between \alpha$;*
3. *α is convergent: $s_1 \mathrel{\epsilon} \alpha\,,\ s_2 \mathrel{\epsilon} \alpha \Rightarrow \{s_1\} \downarrow \{s_2\} \between \alpha$.*

6.2 The topology of an interaction structure

Interaction structures can be viewed as an "interactive" reading of the notion of *inductively generated topology.*

6.2.1 Basic topology

Recall that if $w = (A, D, n)$ is an interaction structure on S, propositions 4, 5 and 2 guarantee that:

- w^* is a closure operator on the subsets of S;
- $w^{\perp^\infty}$ is an interior operator on the subsets of S.

We also have the execution formula (page 29): $\dfrac{w^*(U) \between w^{\perp^\infty}(V)}{U \between w^{\perp^\infty}(V)}$.

As a result, we put:

Definition 22 *If w is an interaction structure on S, define:*

$$\begin{array}{rcl} \mathcal{A}_w(U) & \triangleq & w^*(U) \ ; \\ \mathcal{J}_w(U) & \triangleq & w^{\perp^\infty}(U) \ . \end{array}$$

We have:

Lemma 6.3 *If $w : S \to \mathit{Fam}^2(S)$, then $(S, \mathcal{A}_w, \mathcal{J}_w)$ is a basic topology.*

In [9], the authors use the notion of axiom set to inductively generate a formal cover. The difference between axiom sets and interaction structures is merely that an axiom set is an element of the type $S \to \mathit{Fam}\big(\mathit{Pow}(S)\big)$ that was mentioned on page 20.

If we look at the rules used to generate $\lhd_w$, *i.e.* for w^*, we obtain:

- $\dfrac{s \epsilon U}{s \epsilon \mathcal{A}(U)}$ EXIT;
- $\dfrac{a \in A(s) \qquad \big(\forall d \in D(s,a)\big) \big(n(s,a,d) \epsilon \mathcal{A}(U)\big)}{s \epsilon \mathcal{A}(U)}$ CALL.

Those correspond exactly to the *reflexivity* and *infinity* rules used in [9] to generate the cover "$\lhd$".

6.2.2 Continuous relations revisited

We argued above that (generated) basic topologies and interaction structures are the same notions with different intuitions. We will now lift the notion of continuity to the realm of interaction structures. The result is that in basic topology, continuous relations are exactly general simulations (proposition 12 and lemma 6.5).

Before anything else, let's prove a little lemma about the $\mathcal{J}$ operator:

Lemma 6.4 *Suppose w_h and w_l are interaction structures, and R is a general simulation of $w_h \to w_l$. Then $R^\sim \cdot \mathcal{J}_l(V) \subseteq \mathcal{J}_h \cdot R^\sim(V)$ for all $V \subseteq S_l$.*

Proof: Suppose that $V \subseteq S_l$, $(s_h, s_l) \epsilon R$ and $s_l \epsilon \mathcal{J}_l(V)$; we need to show that $s_h \epsilon \mathcal{J}_h\big(R^\sim(V)\big)$. Since $\mathcal{J}_h\big(R^\sim(V)\big)$ is the greatest fixpoint of the operator $\big(R^\sim(V)\big) \cap w^\perp(\_)$, it suffices to show that s_h is in a pre-fixpoint of the same operator. We claim that $R^\sim(V)$ is such a pre-fixpoint:

- $s_h \epsilon R^{\sim}(V)$ because $(s_h, s_l) \epsilon R$ and $s_l \epsilon \mathcal{J}_l(V) \subseteq V$;
- $R^{\sim}(V) \subseteq R^{\sim}(V)$;
- $R^{\sim}(V) \subseteq w^{\perp}\big(R^{\sim}(V)\big)$:

  let $s_h \in R^{\sim}(V)$ and $a_h \in A_h(s_h)$; we need to find a $d_h \in D_h(s_h, a_h)$ s.t. $s_h[a_h/d_h] \epsilon R^{\sim}(V)$.
  By lemma 5.3, we know that $s_l \epsilon \mathcal{A}_l\big(\bigcup_{d_h} R(s_h[a_h/d_h])\big)$, and because $s_l \epsilon \mathcal{J}_l(V)$, we can apply the execution formula to obtain a final state $s'_l \epsilon \bigcup_{d_h} R(s_h[a_h/d_h]) \cap \mathcal{J}_l(V)$. In particular, there is a $d_h \in D_h(s_h, a_h)$ such that $s'_l \epsilon R(s_h[a_h/d_h])$.
  Since $s'_l \epsilon \mathcal{J}_l(V) \subseteq V$, it implies that $s_h[a_h/d_h] \epsilon R^{\sim}(V)$.

□

With this new lemma, it is easy to prove the following:

Proposition 12 *Let w_h and w_l be two interaction structures, let R be a relation between S_h and S_l; R is a general simulation $w_h \rightarrow w_l$ iff $R^{\sim}$ is a continuous relation from $(S_l, \mathcal{A}_l, \mathcal{J}_l)$ to $(S_h, \mathcal{A}_h, \mathcal{J}_h)$.*

Proof: Suppose first that $R^{\sim}$ is continuous; the definition implies in particular that $R\big(\mathcal{A}_h(U)\big) \subseteq \mathcal{A}_l\big(R(U)\big)$ for all $U \subseteq S_h$. By lemma 5.3, this implies that R is a general simulation from w_h to w_l.

The converse is a direct application of lemma 5.3 and lemma 6.4. □

The category of basic topologies and continuous relation **BFTop** has a notion of equality which is more subtle (though coarser) than plain extensional equality of relations. Transposing it in our context we get: $R \approx Q$ if and only if $\mathcal{A}\big(R(s_h)\big) = \mathcal{A}\big(Q(s_h)\big)$ for all $s_h \in S_h$.

Lemma 6.5 *If R and Q are simulations, then R and Q are topologically equal iff their saturations are extensionally equal. (R and Q have the same potential, see page 37.)*

6.2.3 Topological product

In section 4.6, we introduced a notion of binary "angelic tensor", morally corresponding to the union of several interaction structures. This operation was already defined in [9] (and probably in other places) as the product topology. In particular, we have the two continuous projection relations.

Lemma 6.6 *If w_1 and w_2 are two interaction structures, then the two following relations*

- $\pi_1 = \{(s_1, (s_1, s_2)) \in S_1 \times (S_1 \times S_2)\}$;
- $\pi_2 = \{(s_2, (s_1, s_2)) \in S_2 \times (S_1 \times S_2)\}$

are (linear) simulations from w_i to $w_1 \odot w_2$ (for $i = 1, 2$); and a fortiori are morphisms in all the categories considered.

If one interprets the sets S_1 and S_2 as (pre)bases, and $\lhd$ are the covering relation; it is clear that this corresponds indeed to the usual product of topologies.

To make this statement precise would require a deeper analysis of continuous relations in the context of convergent topologies.[12]

6.2.4 Extending the execution formula

The definition of basic topology places few constraints on the $\mathcal{A}$ and $\mathcal{J}$ operators. Compatibility is a very weak requirement. On the other hand, the $\mathcal{A}_w$ and $\mathcal{J}_w$ generated from an interaction structure w have a lot in common. In particular, *classically*, $\mathcal{A}_w$ and $\mathcal{J}_w$ are dual: $\complement \fatsemi \mathcal{A}_w = \mathcal{J}_w \fatsemi \complement$; and the positivity axiom is classically always true! It is thus natural to ask whether we can extend our interpretation to take into account more basic topologies. It is possible if we use different interaction structures to generate the $\mathcal{A}$ and the $\mathcal{J}$:

Proposition 13 *Suppose that R is a simulation of (S_h, w_h) by (S_l, w_l). Then*

1. *$\langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim]$ is an interior operators on $Pow(S_h)$;*
2. *$\mathcal{A}_h$ is compatible with $\langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim]$.*

i.e. $\big(S_h\, , \mathcal{A}_h\, , \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim]\big)$ is a basic topology.

Proof: First point: $\langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim]$ is an interior operator:

- $\langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](U) \subseteq \langle R\rangle \cdot [R^\sim](U) \subseteq U$
- we have:

$$\begin{array}{ll}
 & \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](U) \subseteq V \\
\Rightarrow & \\
 & [R^\sim] \cdot \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](U) \subseteq [R^\sim](V) \\
\Rightarrow & \ \big\{ \text{ since } U \subseteq [R^\sim] \cdot \langle R\rangle(U) \ \big\} \\
 & \mathcal{J}_l \cdot [R^\sim](U) \subseteq [R^\sim](V) \\
\Rightarrow & \ \big\{ \ \mathcal{J}_l \text{ is an interior operator } \big\} \\
 & \mathcal{J}_l \cdot [R^\sim](U) \subseteq \mathcal{J}_l \cdot [R^\sim](V) \\
\Rightarrow & \\
 & \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](U) \subseteq \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](V)
\end{array}$$

This completes the proof that $R^\sim \cdot \mathcal{J}_l \cdot [R^\sim]$ is an interior operator.

Second point: $\langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim]$ is compatible with $\mathcal{A}_h$.
Let $s_h \epsilon \mathcal{A}_h(U)$ and $s_h \epsilon \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](V)$, *i.e.* we have an s'_l s.t. $(s_h, s'_l) \epsilon R$ and $s'_l \epsilon \mathcal{J}_l \cdot [R^\sim](V)$. In particular, $s'_l \epsilon R\big(\mathcal{A}_h(U)\big)$ and so $s'_l \epsilon \mathcal{A}_l\big(R(U)\big)$ by lemma 5.3.
We can apply the execution formula in w_l to obtain a final state $s''_l \epsilon \langle R\rangle(U)$ s.t. $s''_l \epsilon \mathcal{J}_l \cdot [R^\sim](V)$, *i.e.* there is an $s'_h \epsilon U$ s.t. $(s'_h, s''_l) \epsilon R$, which implies that $s'_h \epsilon \langle R\rangle \cdot \mathcal{J}_l \cdot [R^\sim](V)$.

□

An interactive reading is that for interaction to take place, the Angel and Demon do not need to use exactly same dialect. If the Angel uses w_h and the Demon uses w_l, the Demon needs to interpret actions in w_h in terms of actions in w_l, and the Angel needs to interpret reactions in w_l in terms of reactions in

[12] *i.e.* one wants to prove that $\odot$ is a cartesian product in the category of "localized interaction structures" (see definition 24 on page 46) with "convergent and total" general (see [15]) simulations.

w_h, *i.e.* we need to have a simulation from w_h to w_l. Note that because of the respective roles of the Angel and Demon, one never needs to translate actions from the Demon or reaction from the Angel.

In [42], Silvio Valentini investigates the problem of "completeness" of inductively generated topologies. It might be interesting to investigate the operation described above in this context.

6.3 Localization and distributivity

The basic topology obtained from an interaction structure is not in general distributive. One way to obtain distributivity is to add a condition of *convergence* (page 40). In [9], the authors introduce the notion of "localized" axiom set which gives rise to convergent basic topologies, *i.e.* formal topologies.

If $w = (A, D, n)$ is an interaction structure, a preorder $\leq$ on S is said to be localized if the following holds:

$$s' \leq s \ , \ a \in A(s) \quad \Rightarrow \quad s' \mathrel{\epsilon} w\left(\bigcup_{d \in D(s,a)} \{s[a/d]\} \downarrow \{s'\} \right)$$

This implies in particular that $\geq$ is a linear simulation.

Suppose that $\leq$ is localized; if we extend the generating rules with

$$\frac{s' \mathrel{\epsilon} U \qquad s \leq s'}{s \mathrel{\epsilon} \mathcal{A}(U)} \ \leq\text{-compat.}$$

then the resulting lattice is distributive: this is one of the results of [9]. (The rules were slightly more complex because they had to consider the positivity predicate and the positivity axiom). Note that since $\leq$ is a preorder —and as such, reflexive— this rule is a generalization of the reflexivity rule.

The preorder is intended to represent *a priori* the notion of inclusion between basic opens. The smallest interesting preorder to consider is the following: "$s \leq s'$ iff $s \mathrel{\epsilon} \mathcal{A}\{s'\}$". This preorder is the saturation of the identity and it appears implicitly in the definition of convergence.

The rest of this section is devoted to an analysis of the notion of localization in the context of interaction structures, together with a tentative computational interpretation. It culminates with an interpretation of the notion of formal points in terms of server programs.

6.3.1 Interaction structure with self-simulation

The first step is to add a preorder on states, and to require it to be well-behaved with respect to its parent interaction structure.

Definition 23 *An interaction structure with* self-simulation *on S is a pair (w, R) where*

- *w is an interaction structure on S;*
- *R is a general simulation from w to itself.*

Lemma 6.7 *If R is a simulation from w to itself, then so is the reflexive transitive closure of R.*

Proof: This is a direct consequence of the following facts: identities are simulations, simulations compose (proposition 8) and simulations are closed under unions (proposition 7).

□

As a result, without loss of generality we can assume the self-simulation to be a preorder, and we call it "$\geq$", with converse $\leq$. The meaning of "$s \leq s'$" is thus "s simulates s' in w". We write $\{s\}^{\downarrow}$ for the segment $(s \geq)$ (or $(\leq s)$) below $s \in S$, and $U^{\downarrow}$ for the downclosure $\langle\leq\rangle(U)$ of $U : Pow(S)$.
We have:

Lemma 6.8 *$s \lhd V$ implies $\{s\}^{\downarrow} \lhd V^{\downarrow}$.*

Proof: This is just an application of proposition 6. □

Two extreme examples of such self simulations are:

- the empty relation, or the identity (its reflexive/transitive closure). This is isomorphic to the case of normal interaction structures.
- $R \triangleq (\mathcal{J}_w(S) \times S) \cup (S \times \mathcal{A}_w(\emptyset))$. The intuition is that $(s_D, s_A) \epsilon R$ iff the Demon can avoid deadlocks from s_D or the Angel can deadlock the Demon from s_A.[13] Classically, this can be shown to be the biggest simulation (*i.e.* it is the union of all simulations) on an interaction structure and *a fortiori*, we have $R = R^* = \overline{R^*}$.

Remark. The second example can be seen as a constructive contrapositive of the following fact:
Lemma 6.9 *Let $\geq$ be a self-simulation on an interaction structure (A, D, n) on S; suppose that $s \leq s'$ (s simulates s'); we have:*

- *if the Demon can avoid deadlocks from s' then he can also avoid deadlocks from s (i.e. $s' \ltimes S \Rightarrow s \ltimes S$);*
- *if the Angel can drive the Demon into a deadlock from s', then she can do it from s (i.e. $s' \lhd \emptyset \Rightarrow s \lhd \emptyset$).*

Classically, the two points are equivalent.
The second simulation is (classically) equivalent to the one *defined* from those properties (*i.e.* $(s, s') \epsilon R$ iff $s' \lhd \emptyset \Rightarrow s \lhd \emptyset$ iff $s' \ltimes S \Rightarrow s \ltimes S$).

6.3.2 Interaction structures and localization

We now investigate the result of strengthening the condition to get full localization.

Definition 24 *Let $(w, \geq)$ be an interaction structure with self simulation on S; we say that $(w, \geq)$ is localized if the following holds:*

$$s_1 \leq s_2\ ,\ a_2 \in A(s_2) \quad \Rightarrow \quad s_1 \lhd_w \bigcup_{d_2 \in D(s_2,a_2)} \{s_2[a_2/d_2]\} \downarrow \{s_1\}\ .$$

[13] A Demon deadlock is a pair (s, a) such that $D(s, a) = \emptyset$.

This condition is slightly more general than the one from [9] in the sense that it considers general simulations rather than linear ones. Note also that in contrast to the notion of convergence from definition 20, this definition doesn't require equality: $\{s\}^{\downarrow}$ is defined in terms of $\leq$.

First, we need a lemma.

Lemma 6.10 *Suppose* $(w, \geq)$ *is localized on* w*; then we have:*

$$s_1 \leq s_2\ ,\ a'_2 \in A^*(s_2) \quad \Rightarrow \quad s_1 \lhd_w \bigcup_{d'_2 \in D^*(s_2, a'_2)} \{s_2[a'_2/d'_2]\} \downarrow \{s_1\}\ .$$

This means that the additional condition is well behaved with respect to the RTC operation. (This is analogous to point *3* of lemma 5.3; and indeed, the proof is very similar.)

Proof: Suppose that $s_1 \leq s_2$ and let $a'_2 \in A^*(s_2)$. We work by induction on the structure of a'_2.

- if $a'_2 = \textsc{exit}$, this is trivial.
- if $a'_2 = \textsc{call}(a_2, k_2)$: by localization, we know that

  $$s_1 \lhd \bigcup_{d_2} \{s_2[a_2/d_2]\} \downarrow \{s_1\}\ .$$

  Let $s'_1\ \epsilon \bigcup_{d_2} \{s_2[a_2/d_2]\} \downarrow \{s_1\}$, in particular $s'_1 \leq s_2[a_2/d_2]$ for some $d_2 \in D_2(s_2, a_2)$. We can apply the induction hypothesis for $s'_1 \leq s_2[a_2/d_2]$ and $k_2(d_2)$ to obtain:

  $$s'_1 \quad \lhd \bigcup_{d'_2 \in D^*(s_2[a_2/d_2], k_2(d_2))} \big\{s_2[a_2/d_2]\big[k_2(d_2)/d'_2\big]\big\} \downarrow \{s'_1\}\ .$$

  We have $\bigcup_{d'_2 \in D^*(s_2[a_2/d_2], k_2(d_2))} \subseteq \bigcup_{d'_2 \in D^*(s_2, a'_2)}$ and $\{s'_1\}^{\downarrow} \subseteq \{s_1\}^{\downarrow}$ (because $s'_1 \leq s_1$), which implies that the right hand side is thus included in $\bigcup_{d'_2 \in D^*(s_2, a'_2)} \{s_2[a'_2/d'_2]\} \downarrow \{s_1\}$. By monotonicity, we get

  $$\bigcup_{d_2} \{s_2[a_2/d_2]\} \downarrow \{s_1\} \quad \lhd \bigcup_{d'_2 \in D^*(s_2, a'_2)} \{s_2[a'_2/d'_2]\} \downarrow \{s_1\}\ .$$

  We get the result by transitivity.

□

Lemma 6.11 *If* $(w, \geq)$ *is a localized interaction structure, then* $s \lhd U$ *implies* $s \lhd U \downarrow \{s\}$.

Proof: Let $s \lhd U$, *i.e.* there is a a' in $A^*(s)$ s.t. $\bigcup_{d' \in D^*(s, a')} \{s[a'/d']\} \subseteq U$. Since $\leq$ is reflexive, and by the previous lemma, we know that

$$s \lhd \bigcup_{d'} \{s[a'/d']\} \downarrow \{s\}\ .$$

The RHS is obviously included in $U \downarrow \{s\}$, so we get the result by monotonicity.

□

Corollary 4 *If $(w, \geq)$ is a localized interaction structure, then $U \lhd V$ implies $U \lhd U \downarrow V$.*

We will now check that convergence is satisfied for such a $(S, \mathcal{A}_w, \leq)$.

Proposition 14 *If $(w, \geq)$ is a localized interaction structure, then $s \lhd U$ and $s \lhd V$ jointly imply $s \lhd U \downarrow V$.*

Proof: By lemma 6.11, we know that $s \lhd U \downarrow \{s\}$. By lemma 6.8 we know that $\{s\}^{\downarrow} \lhd V^{\downarrow}$, which implies $U \downarrow \{s\} \lhd V^{\downarrow}$; which (by corollary 4) implies $(U \downarrow \{s\})^{\downarrow} \lhd V \downarrow (U \downarrow \{s\})$.

Since we have $(U \downarrow V)^{\downarrow} = U \downarrow V$, we can deduce that

$$s \quad \lhd \quad U \downarrow \{s\} \quad \lhd \quad U \downarrow V \downarrow \{s\} \quad \subseteq \quad U \downarrow V \ .$$

□

However, strictly speaking, the proof of lemma 6.2 doesn't apply to the preorder context. Instead, we have to match the operator generated by adding the $\leq$-compat rule (on page 45), and define:

Definition 25 *if $(w, \leq)$ is an interaction structure with self simulation, we write $\mathcal{A}_{w,\leq}$ for the predicate transformer $U \mapsto \mathcal{A}_w(U^{\downarrow})$, where $U^{\downarrow}$ is the down-closure of U, i.e. $U^{\downarrow} \triangleq \{s \mid (\exists s' \epsilon U)\, s \leq s'\}$.*

The intuition is quite straightforward: if s' can simulate a state s ($s' \leq s$) in U, then s' is "virtually" in U as well. This is way to say that our notion of simulation is semantically a real simulation.

Corollary 5 *If w is an interaction structure and $\leq$ a localized preorder on S; then the collection of open sets (i.e. the collection of U s.t. $U = \mathcal{A}_{w,\leq}(U)$) is distributive.*

6.3.3 Computational interpretation

In the last section we merely transposed the definitions from [9] to the context of interaction structures. It is not however obvious how to make computational sense of these definitions. We now present an analysis of the localization in computational terms. The key idea is that to interpret localization, one needs to adopt the perspective of the Demon (server, $\ltimes$ operator), rather than that of the Angel (client, $\lhd$ operator).

For example, the computational content of lemma 6.11 is that it is possible for the Angel to conduct interaction in such a way that the behavior of the starting state can always be recovered by simulation. The Angel takes care that she can at any point change her mind, and abandon the current computation. An example of a command which one would *not* have in such a system is "`Reset`", a command to brings the whole system back to factory settings. This would lose all information about previous interactions, which is not possible in a localized structure.

Thus, localization is a strong condition on interaction structures, requiring them to be exceptionally well behaved.

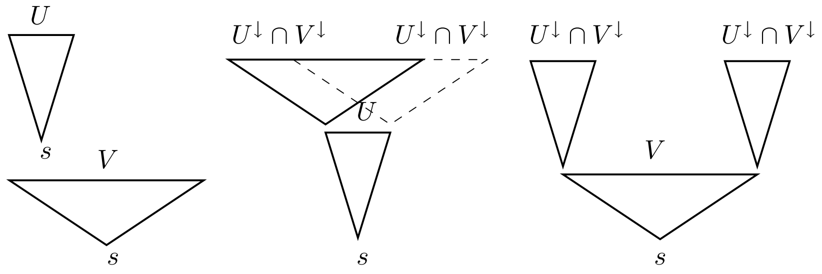
Remark. Note that the notion of "localization" for a game has little to do with the notion of backtracking present in [7], where a game-theoretical interpretation of classical logic is presented.

That the Angel is allowed to backtrack means that the Angel can "go back in the past". If the game is localized, then the Angel does not return to a previous state, but plays in the current state "as if it were" the previous state. In particular, the Angel retains the right to make moves in the current state.

There is a problem with interpreting the proof of proposition 14: a non-canonical choice was made. In the proof, we decided to first execute the client corresponding to $s \lhd U$, and then the client corresponding to $s \lhd V$ on top of it. The opposite works just as well:

$$s \quad \lhd \quad V \downarrow \{s\} \quad \lhd \quad V \downarrow U \downarrow \{s\} \quad \subseteq \quad V \downarrow U\ .$$

The two different witnesses for $s \lhd U \downarrow V$ may be quite different in terms of execution! Here is what happens in graphical terms:


U
s
V
s
$U^{\downarrow} \cap V^{\downarrow}$
$U^{\downarrow} \cap V^{\downarrow}$
U
s
$U^{\downarrow} \cap V^{\downarrow}$
$U^{\downarrow} \cap V^{\downarrow}$
V
s

On the left are the two client programs witnessing $s \lhd U$ and $s \lhd V$; and on the right, the two different programs witnessing $s \lhd U \downarrow V$.

Even worse, when the programs corresponding to $s \lhd U$ and $s \lhd V$ are non-trivial, we could interleave the programs before reaching $U \downarrow V$!

To give computational sense to the notion of localization, consider a server interacting with clients. We allow ourselves a degree of anthropomorphism, by referring to what these parties "believe".

Think of the self-simulation $\geq$ as a relation between "visible" or "virtual" states for the client(s) and "internal" server states. Because this is a (general) simulation, it is guaranteed that we can conduct interaction in the following way:

1. if $s' \leq s$, *i.e.* the Angel believes the Demon is in a state s, but internally, the Demon is really in a state s' that simulates s;
2. the Angel sends a request $a \in A(s)$;
3. the Demon does the following:
   (a) translates the $a \in A(s)$ into a $a' \in A^*(s')$ (by simulation),
   (b) responds to a' with a $d' \in D^*(s', a')$ (because it is a server program),
   (c) and translate this answer d' into a $d \in D(s, a)$ (by simulation);
4. The Angel receives the answer $d \in D(s, a)$;

5. the Angel now believes the new state is $n(s,a,d)$ while internally, the Demon is really in state $n^*(s',a',d') \leq n(s,a,d)$ that simulates $n(s,a,d)$;

In particular, after the last point, the Angel can continue interaction.

Localization can then be seen as the following requirement: suppose the server is internally in a state s and that there are two clients who respectively believe it is in state s_1 and s_2. The two clients can send their requests and the server respond to them (as above) in any order. Suppose the server first responds the first client. Then at point (a) in the analysis of client-server interaction above, the server can chose some a' which is constrained to bring about a state $s'[a'/d'] \leq s_1[a_1/d_1]$ (like above) *and* $s'[a'/d'] \leq s$ (by localization). The first condition allows the first client to continue interaction, while the second point (localization) guarantees that the server can also answer requests to the second client (because $s'[a'/d'] \leq s \leq s_2$)...

In other words, that an interaction structure is equipped with a localized self-simulation means that we can construct "concurrent virtual servers" with which several clients can interact independently.

One way to localize any interaction structure on S is the following: define $\text{Ł}(w)$ on $\mathit{Fin}(S)$[14] as

$$\begin{array}{lcl}
\text{Ł}(w).A(\{\, s_i \mid i \in I \,\}) & \triangleq & (\Sigma\, i \in I)\, w.A(s_i) \\
\text{Ł}(w).D(\{\, s_i \mid i \in I \,\}, (i,a)) & \triangleq & w.D(s_i, a) \\
\text{Ł}(w).n(\{\, s_i \mid i \in I \,\}, (i,a), d) & \triangleq & \{\, s_i \mid i \in I \,\} \cup \{s[a/d]\}
\end{array}$$

with reverse inclusion as simulation order. (To define the inclusion order between families of states of course requires there to be an equality relation between states.) This interaction structure $(\text{Ł}(w), \geq)$ is automatically localized.

The idea is simply that the Demon keeps a log of all the previous states visited during interaction, so that he can use any "past" state as the current one.

> **Remark.** To get a situation which is even closer to "real life", one can define a simulation $R : w \rightarrow \text{Ł}(w)$ with $(s,l) \mathrel{\epsilon} R$ iff "$s \mathrel{\epsilon} l$" and use the extension from section 6.2.4 to interpret interaction.
>
> The idea is that the Demon advertizes a service specified by the interaction structure w; but internally implements $\text{Ł}(w)$ in order to deal with concurent requests. The clients are only supposed to use interface w.

Points. Now that the notion of localized interaction structure (aka formal topology) has a computational interpretation, we can look at the notion of formal point. Recall that a formal point in a formal topology on S is a subset $\alpha \subseteq S$ such that (see [15])

- α is closed;
- α is non-empty ($\alpha \between \alpha$);
- α is convergent, *i.e.* $s \mathrel{\epsilon} \alpha$ and $s' \mathrel{\epsilon} \alpha$ imply that $s \downarrow s' \between \alpha$.

[14] where $\mathit{Fin}(S)$ is the collection of finite subsets of S. Finite subsets are represented by families indexed by a finite set (*i.e.* an integer).

We have already described the computational interpretation of a closed subset in section 4.5: the closed subset $\mathcal{J}(V)$ is a specification for a server program that can maintain V.[15] That it is non-empty means that we actually have a proof that $s \ltimes \alpha$ for some s, *i.e.* that we have a server program maintaining V (from some specific state s).

Thus, a point is nothing more than a specification for a server program that satisfies

- if $s_1 \epsilon \alpha$ (a client may connect in state s_1)
- and if $s_2 \epsilon \alpha$ (a client may connect in state s_2)
- then there is a(n internal) state s that simulates both s_1 and s_2 (since $s \epsilon s_1 \downarrow s_2$) such that $s \epsilon \alpha$. In other words, the server can find an internal state which will allow it to respond to both s_1 and s_2.

Formal points are thus "coherent" server program specifications in the sense that they can satisfy any finite number of "unrelated" concurrent clients.

Continuous maps. A relation R between two localized interaction structures w_h and w_l is called a continuous map if we have (see [15]):

- R is a general simulation from w_h to w_l;
- R is total: $S_l \lhd_{w_l,\leq} R(S_h)$;
- $R(s_1) \downarrow R(s_2) \lhd_{w_l,\leq} R(s_1 \downarrow s_2)$.

Similar interpretation can be devised for continuous maps as for formal points, but the relevance of this interpretation in terms of actual client/server programming is still unclear. We prefer to leave the matter open for the time being.

7 Conclusion, and questions raised

We hope to have shown that much of *basic* topology has a natural interpretation in programming terms. On reflection this is not surprising: programming is essentially about "how to get there from here", and this is a notion with a topological flavor.

Our study of interaction structures began with the intention of clarifying monotone predicate transformers such as those which model specifications in imperative programming. We have defined a category in which the objects represent command-response interfaces, and the morphisms represent program components that implement one (higher-level) interface "on top of" other (lower-level) interfaces. The category coincides with the category of basic spaces and continuous relations. Closure and interior operators are related to server programs and client programs, and continuous maps to simulations of one server "on top of" another. We have tentatively proposed a computational interpretation of those notions of formal topology connected with convergence, and particularly the notion of point.

[15] It is straightforward to extend the $\mathcal{J}$ operator to the case of interaction structures with self simulations: $\mathcal{J}_{w,\leq}(V) = \mathcal{J}_w(V^\uparrow)$.

We would also like to find topological counterparts of fundamental computational notions. For example, safety properties are essentially the same as closed sets; but what about fairness properties? For another example we have seen that the notion of forward data refinement in programming is connected with the notion of continuity (at least at the level of basic topology). From the computer science literature, it is known that both forward data refinement and backward data refinement are required for refinement of abstract data types (see for example [13]). Similar completeness properties hold in approaches to refinement based on functions and auxiliary variables rather than relations (see for example the use of history and prophecy variables as in [1]). It seems interesting therefore to ask whether backward simulation or the use of prophecy variables has a topological interpretation.

Another line of work concerns the model of classical linear logic presented in [23].

Finally, one hopes that the use of *dependent* theory type permits the expression of interface specifications with full precision —that is, going beyond mere interface signatures. This might serve as a foundation for designing components in real programming languages. Tools to aid design might be built on this foundation. However examples of interfaces and simulations are needed both to ensure that our model properly captures important properties of interfaces, and also to find ergonomically smooth ways of working with simulations.

Acknowledgments. Thierry Coquand drew the authors' attention to the Peterson and Synek tree sets and explained their connection with inductively defined coverage systems in formal topology. He stressed the importance of simulation. He organized a meeting with Sambin. Both authors are indebted to him. Sambin's work on formal topology has been a delightful source of plunder. The second author also wants to thank Giovanni Sambin for inviting him twice: once for the Second Workshop on formal topology in Venice; and once on a personal basis in Padova to share ideas. The collaboration of Anton Setzer has helped the first author's understanding of coinductive notions ([19, 30], and of coalgebraic programming with dependent types ([17, 18]).

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2, Logic Semantics Theory Program.):253–284, 1991.

[2] Peter Aczel. An introduction to inductive definitions. In *Handbook of mathematical logic*, pages 739–782, Amsterdam, 1977. North-Holland Publishing Co. Edited by Jon Barwise, With the cooperation of H. J. Keisler, K. Kunen, Y. N. Moschovakis and A. S. Troelstra, Studies in Logic and the Foundations of Mathematics, Vol. 90.

[3] Bowen Alpern and Fred Barry Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement calculus, A systematic introduction.* Graduate Texts in Computer Science. Springer-Verlag, New York, 1998.

[5] Ralph-Johan Back and Joakim von Wright. Products in the refinement calculus. Technical Report 235, Turku Center for Computer Science, February 1999.

[6] Ralph-Johan Back and Joakim Von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12(5):313–349, 2000.

[7] Thierry Coquand. A semantics of evidence for classical arithmetic. *The Journal of Symbolic Logic*, 60(1):325–337, 1995.

[8] Thierry Coquand and Catarina Coquand. The Agda proof assistant. `http://www.cs.chalmers.se/ catarina/agda/`, 2000.

[9] Thierry Coquand, Giovanni Sambin, Jan Smith, and Silvio Valentini. Inductively generated formal topologies. *Annals of Pure and Applied Logic*, 124(1-3):71–106, 2003.

[10] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[11] Albert Grigorevich Drágalin. A completeness theorem for intuitionistic predicate logic. An intuitionistic proof. *Publicationes Mathematicae Debrecen*, 34(1-2):1–19, 1987.

[12] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[13] Paul H. B. Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[14] Silvia Gebellato and Giovanni Sambin. The essence of continuity (the Basic Picture, II), 2001. Preprint n. 27, Dipartimiento di matematica, Università di Padova.

[15] Silvia Gebellato and Giovanni Sambin. Pointfree continutity and convergence (the Basic Picture, IV), 2002. draft.

[16] Andrzej Grzegorczyk. A philosophically plausible formal interpretation of intuitionistic logic. *Indagationes Mathematicae, Koninklijke Nederlandse Akadademie van Wetenschappen, Proceedings Series A 67*, 26:596–601, 1964.

[17] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Computer science logic (Fischbachau, 2000)*, volume 1862 of *Lecture Notes in Comput. Sci.*, pages 317–331. Springer, Berlin, 2000.

[18] Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, July 7th 2000*, 2000.

[19] Peter Hancock and Anton Setzer. Guarded induction and weakly final coalgebras in dependent type theory. 15 pp., submitted, available via `http://www.cs.swan.ac.uk/ csetzer/`, 2004.

[20] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[21] Martin Hofmann. Elimination of extensionality and quotient types in martin-löf's type theory. In *Types for Proofs and Programs, International Workshop TYPES'93*, volume 806 of *LNCS*. Springer Verlag, 1994.

[22] Pierre Hyvernat. Predicate transformers and linear logic: second order. Unpublished, `http://iml.univ-mrs.fr/ hyvernat/academics.html`, 2004.

[23] Pierre Hyvernat. Predicate transformers and linear logic: yet another denotational model. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Workshop CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag, September 2004.

[24] Pierre Hyvernat. Synchronous games, linear logic and the differential λ-calculus. Unpublished, 2004.

[25] Peter T. Johnstone. The point of pointless topology. *Bulletin of the American Mathematical Society*, 8(1):41–53, 1983.

[26] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[27] Leslie Lamport. Formal foundation for specification and verification. In Alford et al., editor, *Distributed Systems: Methods and Tools for Specification*, number 190 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1985.

[28] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[29] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[30] Markus Michelbrink and Anton Setzer. State-dependent IO-monads in type theory. submitted, available via `http://www.cs.swan.ac.uk/ csetzer/`, 2004.

[31] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[32] Bengt Nordström, Kent Petersson, and Jan Magnus Smith. *Programming in Martin-Löf's type theory, an introduction*. The Clarendon Press Oxford University Press, New York, 1990.

[33] Kent Petersson and Dan Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, UK*, volume 389 of *LNCS*. Springer Verlag, 1989.

[34] Giovanni Sambin. Intuitionistic formal spaces—a first communication. In *Mathematical logic and its applications (Druzhba, 1986)*, pages 187–204. Plenum, New York, 1987.

[35] Giovanni Sambin. The Basic Picture, a structure for topology (the Basic Picture, I), 2001. Preprint n. 26, Dipartimiento di matematica, Università di Padova.

[36] Giovanni Sambin. Basic topologies, formal topologies, formal spaces (the Basic Picture, III), 2002. draft.

[37] Giovanni Sambin. Some points in formal topology. *Theoretical Computer Science*, 305(1-3):347–408, 2003. Topology in computer science (Schloß Dagstuhl, 2000).

[38] Giovanni Sambin and Silvia Gebellato. A preview of the basic picture: a new perspective on formal topology. In *Types for proofs and programs (Irsee, 1998)*, pages 194–207. Springer, Berlin, 1999.

[39] Giovanni Sambin and Silvio Valentini. Building up a toolbox for Martin-Löf's type theory: subset theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, pages 221–244. Oxford Univ. Press, New York, 1998.

[40] Fred Barry Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[41] Alfred Tarski. Fundamentale Begriffe der Methodologie der deduktiven Wissenschaften. I. *Monatshefte für Mathematik und Physik*, 37:361–404, 1930.

[42] Silvio Valentini. The problem of completeness of formal topologies with a binary positivity predicate and their inductive generation, 2004. To be published in Archive for Mathematical Logic.

[43] Steven Vickers. *Topology via logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.